

Android中高级面试题汇总 (2022年)

第一章 Java基础

第一节 静态内部类和非静态内部类的比较

1.1 静态内部类和非静态内部类的区别

静态内部类只能访问外部类的静态成员和静态方法

非静态内部类不管是静态方法还是非静态方法都可以在非静态内部类中访问

静态内部类和非静态内部类主要的不同：

- (1) 静态内部类不依赖于外部类实例而被实例化，但非静态内部类需要在外部类实例化后才可以被实例化
- (2) 静态内部类不需要持有外部类的引用。但非静态内部类需要持有对外部类的引用
- (3) 静态内部类不能访问外部类的非静态成员变量和非静态方法。他只能访问外部类的静态成员和静态方法，非静态内部类能够访问外部类的静态和非静态成员和方法

1.2 扩展：内部类都有哪些？

有四种：**静态内部类、非静态内部类、局部内部类、匿名内部类**

1.3 局部内部类

在外部类的方法中定义的类，其作用的范围是所在的方法内。他不能被 public、private、protected 来修饰。他只能访问方法中定义的 final 类型的局部变量。

1.4匿名内部类：是一种没有类名的内部类。

需要注意的是：

- 1、匿名内部类一定是在new的后面，这个匿名内部类必须继承一个父类或实现一个接口
- 2、匿名内部类不能有构造函数
- 3、只能创建匿名内部类的一个实例
- 4、在Java8之前，如果匿名内部类需要访问外部类的局部变量，则必须用final修饰外部类的局部变量。在现在Java8已经取消了这个限制。

第二节 多态的理解与应用

2.1多态概述

1. 多态是继封装、继承之后，面向对象的第三大特性。

2. 多态现实意义理解：

- 现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三既是**学生**也是**人**，即出现**两种形态**。
- Java作为面向对象的语言，同样可以描述一个事物的多种形态。如 Student类继承了Person类，一个Student的对象便既是Student，又是Person。

3.多态体现为父类引用变量可以指向子类对象。

4.前提条件：必须有子父类关系。

注意：在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

5.多态的定义与使用格式

定义格式：父类类型 变量名=new 子类类型();

6.理解：

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作。

2.2多态中成员的特点

1. 多态成员变量：编译运行看左边

```
Fu f=new Zi();
```

System.out.println(f.num); //f是Fu中的值，只能取到父中的值

2. 多态成员方法：编译看左边，运行看右边

```
Fu f1=new Zi();
```

System.out.println(f1.show()); //f1的门面类型是Fu,但实际类型是Zi,所以调用的是重写后的方法。

2.3instanceof关键字

作用：用来判断某个对象是否属于某种数据类型。

- 注意：返回类型为布尔类型

使用案例：

```
Fu f1=new Zi();
Fu f2=new Son();
if(f1 instanceof Zi){
    System.out.println("f1是Zi的类型");
}
else{
    System.out.println("f1是Son的类型");
}
```

2.4多态的转型

- 多态的转型分为向上转型和向下转型两种

- 向上转型：多态本身就是向上转型过的过程

使用格式：父类类型 变量名=new 子类类型();

适用场景：当不需要面对子类类型时，通过提高扩展性，或者使用父类的功能就能完成相应的操作。

- 向下转型：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用类型转为子类引用各类型

使用格式：子类类型 变量名= (子类类型) 父类类型的变量；

适用场景：当要使用子类特有功能时。

2.5多态案例：

例1：（理解多态，可以重点看这个案例）

```
package day0524;

public class demo04 {
    public static void main(String[] args) {
        People p=new Stu();
        p.eat();
        //调用特有的方法
        Stu s=(Stu)p;
        s.study();
        //((Stu) p).study();
    }
}
class People{
    public void eat(){
        System.out.println("吃饭");
    }
}
class Stu extends People{
    @Override
    public void eat(){
        System.out.println("吃水煮肉片");
    }
    public void study(){
        System.out.println("好好学习");
    }
}
```

```
class Teachers extends People{  
    @Override  
    public void eat(){  
        System.out.println("吃樱桃");  
    }  
    public void teach(){  
        System.out.println("认真授课");  
    }  
}
```

例2：请回答题目运行结果是什么？

```
package day0524;  
public class demo1 {  
    public static void main(String[] args) {  
        A a=new A();  
        a.show();  
        B b=new B();  
        b.show();  
    }  
}  
class A{  
    public void show(){  
        show2();  
    }  
    public void show2(){  
        System.out.println("A");  
    }  
}  
class B extends A{  
    public void show2(){  
        System.out.println("B");  
    }  
}  
class C extends B{  
    public void show(){  
        super.show();  
    }  
}
```

```
public void show2(){  
    System.out.println("C");  
}  
}
```

答案：A B

第三节 java方法的多态性理解

3.1 什么是java的多态

浏览了别人博客中的一些介绍多态的文章，发现大家的描述有点不一样，主要区别在于是否把方法的重写算做多态。一种我比较认同的说法如下：

多态分为两种

- a. 编译时多态：方法的重载；
- b. **运行时多态**：JAVA运行时系统根据调用该方法的实例的类型来决定选择调用哪个方法则被称为运行时多态。（我们平时说得多的事运行时多态，所以多态主要也是指运行时多态）；

上述描述认为重载也是多态的一种表现，不过多态主要指运行时多态。

3.2 运行时多态

- a. **面向对象的三大特性**：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。
- b. **多态的定义**：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）
- c. **实现多态的技术称为**：动态绑定（dynamic binding），是指在**执行期间**判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。
- d. **多态的作用**：消除类型之间的耦合关系。

e. 现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

多态存在的三个必要条件

- 一、要有继承；
- 二、要有重写；
- 三、父类引用指向子类对象。

多态的好处：

1. 可替换性 (substitutability)。多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。

2. 可扩充性 (extensibility)。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

3. 接口性 (interface-ability)。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图8.3 所示。图中超类Shape规定了两个实现多态的接口方法，computeArea()以及computeVolume()。子类，如Circle和Sphere为了实现多态，完善或者覆盖这两个接口方法。

4. 灵活性 (flexibility)。它在应用中体现了灵活多样的操作，提高了使用效率。

5. 简化性 (simplicity)。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

3.3 代码理解

看了上面的描述，我们大概知道以下几点：

a. 运行时多态是在父类引用指向子类对象时产生的。一个父类的引用可以指向多种子类对象，那么运行时对于同一个消息应该如何做出响应呢？这就由实际的被引用的对象的类型来决定。

b. 为什么要有重写呢？难道父类中的方法没有被重写，直接调用子类中存在的方法难道是不行吗？看个例子如下：

```
public class Solution {  
    public class Father{  
        public String name = "father";  
        //  
        //  
        //  
        public String getName(){  
            return this.name;  
        }  
    }  
  
    public class Son extends Father{  
        public String name = "son";  
        public String getName(){  
            return this.name;  
        }  
    }  
  
    public static void main(String arg[]){  
        Solution solution = new Solution();  
        Solution.Father father = solution.new Son();  
        System.out.print(father.getName());  
    }  
}
```

上面的例子中，当父类中的getName()被注释掉以后，调用father.getName()方法会出错。因此，当父类引用指向子类方法时，必须调用那些父类中存在的方法，如果子类中对该方法进行了重写，那么在运行时就会动态调用子类中的方法，这就是多态。

c. 要有继承很好理解，没有继承的话，哪来的重写呢。

3.4深一点

基本了解了多态以后，我们就可以看明白下面这个例子了，它的输出结果是什么呢？

```
public class Solution {  
    public class Father{  
        public String name = "father";  
        public String getName(){  
            return this.name;  
        }  
    }  
  
    public class Son extends Father{  
        public String name = "son";  
        public String getName(){  
            return this.name;  
        }  
    }  
  
    public static void main(String arg[]){  
        Solution solution = new Solution();  
        Solution.Father father = solution.new Son();  
        System.out.print(father.getName());  
    }  
}
```

答案是"son"，结合前面的解释，我们很容易判断出来。子类的getName()方法重写了父类中的方法，在main中，父类的引用father指向了子类的对象son，当我们调用father的getName()方法时，实际上

3.5再深一点

你是否真正理解了多态呢？请看下面的例子：

```
public class A{
    public void show(A obj){
        System.out.println("A and A");
    }
    public void show(C obj){
        System.out.println("A and C");
    }
}
public class B extends A{
    public void show(B obj){
        System.out.println("B and B");
    }
    public void show(A obj){
        System.out.println("B and A");
    }
}
public class C extends B{...}

public static void main(String arg[]){
    Solution solution = new Solution();
    Solution.A a1 = solution.new A();
    Solution.A a2 = solution.new B();
    Solution.B b = solution.new B();
    Solution.C c = solution.new C();
    a1.show(b);a1.show(c);a2.show(b);a2.show(c);
}
```

上面这个例子中，下面四条语句的输出结果是什么呢？

```
a1.show(b);
a1.show(c);
a2.show(b);
a2.show(c);
```

结果如下：

```
A and A
A and C
B and A
A and C
```

对于前两条语句的结果我们很容易理解，那第三条和第四条的，为什么结果和我们想的不一样，不应该是“B and B”吗？要理解这是为什么，我们要先理解下面这句话：

当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。（但是如果强制把超类转换成子类的话，就可以调用子类中新添加而超类没有的方法了）

看一下标红的那句话，我们知道问题所在了吗？

当运行 `a2.show(b)` 的时候，实际是要调用一个 `show(B obj)` 的方法，但是 A 中有这样一个方法吗？没有！但是由于 B 继承自 A，所以会调用 A 中的 `show(A obj)` 的方法，但是调用时候发现这个方法已经被 B 重写了，所以就会转向来调用 B 中的 `show(A obj)` 方法。所以才会打印出 "B and A"。

实际上这里涉及方法调用的优先问题，优先级由高到低依次为：

this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)。让我们来看看它是怎么工作的。

比如，`a2.show(b)`，`a2` 是一个引用变量，类型为 A，则 `this` 为 `a2`，`b` 是 B 的一个实例，于是它到类 A 里面找 `show(B obj)` 方法，没有找到，于是到 A 的 `super`(超类)找，而 A 没有超类，因此转到第三优先级 `this.show((super)O)`，`this` 仍然是 `a2`，这里 O 为 B，`(super)O` 即 `(super)B` 即 A，因此它到类 A 里面找 `show(A obj)` 的方法，类 A 有这个方法，但是由于 `a2` 引用的是类 B 的一个对象，B 覆盖了 A 的 `show(A obj)` 方法，因此最终锁定到类 B 的 `show(A obj)`，输出为 "B and A"。

怎么样？理解了吗？

问题还要继续，现在我们再来看上面的分析过程是怎么体现出红色字体那句话的内涵的。它说：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。还是拿 `a2.show(b)` 来说吧。

`a2` 是一个引用变量，类型为 A，它引用的是 B 的一个对象，因此这句话的意思是由 B 来决定调用的是哪个方法。因此应该调用 B 的 `show(B obj)` 从而输出 "B and B" 才对。但是为什么跟前面的分析得到的结果不相符呢？！问题在于我们不要忽略了蓝色字体的后半部分，那里特别指明：这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。B 里面的

show(B obj)在超类A中有定义吗？没有！那就更谈不上被覆盖了。实际上这句话隐藏了一条信息：它仍然是按照方法调用的优先级来确定的。它在类A中找到了show(A obj)，如果子类B没有覆盖show(A obj)方法，那么它就调用A的show(A obj)（由于B继承A，虽然没有覆盖这个方法，但从超类A那里继承了这个方法，从某种意义上说，还是由B确定调用的方法，只是方法是在A中实现而已）；现在子类B覆盖了show(A obj)，因此它最终锁定到B的show(A obj)。这就是那句话的意义所在，到这里，我们可以清晰的理解Java的多态性了。

3.6最后一个练习！

看下面的例子：

```
class A{
    public void show(){
        show2();
    }
    public void show2(){
        System.out.println("wo");
    }
}
class B extends A{
    public void show2(){
        System.out.println("ai");
    }
}
class C extends B{
    public void show2(){
        System.out.println("ni");
    }
}

public class Solution {
    public static void main(String arg[]){
        A a = new B();
        a.show();
        B b = new C();
        b.show();
    }
}
```

上面例子中的输出是什么呢？答案是： ai ni

有了前一个例子我们就会很容易理解这个例子。在B类中是没有对A中的show方法进行重写，所以当a.show()时调用的是父类中的show方法，父类中show方法调用了show2方法，但是在调用的时候发现show2方法已经被子类重写，因此会调用子类中show2方法，因此输出"ai"。可见，当父类引用指向子类对象的时候，对父类中方法的调用都会绑定到子类中重写后的办法上，如果子类没有对方法进行重写，那么会直接调用父类中的方法，**相当于是直接调用从父类继承的方法。**

还需要注意的一点是：子类在重写父类的方法时，方法的访问权限不能更低！

第四节 java中接口和继承的区别

实际概念区别：

区别1：

不同的修饰符修饰(interface),(extends)

区别2：

在面向对象编程中可以有多继承!但是只支持接口的多继承,不支持'继承'的多继承哦

而继承在java中具有单根性,子类只能继承一个父类

区别3：

在接口中只能定义全局常量,和抽象方法

而在继承中可以定义属性方法,变量,常量等...

区别4：

某个接口被类实现时,在类中一定要实现接口中的抽象方法

而继承想调用那个方法就调用那个方法,毫无压力

接口是：对功能的描述 继承是：什么是一种什么

始终记者：你可以有多个干爹（接口），但只能有一个亲爹（继承）

举例：

如果狗的主人只是希望狗能爬比较低的树，但是不希望它继承尾巴可以倒挂在树上，像猴子那样可以飞檐走壁，以免主人管不住它。

那么狗的主人肯定不会要一只猴子继承的狗。

设计模式更多的强调面向接口。猴子有两个接口，一个是爬树，一个是尾巴倒挂。我现在只需要我的狗爬树，但是不要它尾巴倒挂，那么我只要我的狗实现爬树的接口就行了。同时不会带来像继承猴子来带来的尾巴倒挂的副作用。这就是接口的好处。

第五节 线程池的好处，详解，单例（绝对好记）

前几天公司面试，问了很多线程池的问题，由于也是菜鸟一只本来对线程池就不太熟悉，再加上一紧张脑袋一紧，就GG了，之后可谓是深恶痛极，决定把线程池这边好好的整理一番。

5.1 线程池的好处

线程池是啥子，干啥使它呀，老子线程使得好好的，非得多次一举，哈哈，想必来这里看这篇文章的都对线程池有点了解。那么我来整理整理线程池的好处吧。

5.1.1 线程池的重用

线程的创建和销毁的开销是巨大的，而通过线程池的重用大大减少了这些不必要的开销，当然既然少了这么多消费内存的开销，其线程执行速度也是突飞猛进的提升。

5.1.2 控制线程池的并发数

初学新手可能对并发这个词语比较陌生，特此我也是结合百度百科和必生所学得出最优解释，万万记着并发可跟并行不一样。

并发：在某个时间段内，多个程序都处在执行和执行完毕之间；但在一个时间点上只有一个程序在运行。头脑风暴：老鹰妈妈喂小雏鹰食物，小雏鹰很多，而老鹰只有一张嘴，她需要一个个喂过去，到最后每个小雏鹰都可以吃到，但是在任何一个时间点里只能有一个小雏鹰可以吃到美味的食物。

并行: 在某个时间段里，每个程序按照自己独立异步的速度执行，程序之间互不干扰。头脑风暴：这就好似老鹰妈妈决定这样喂食太费劲于是为每个小雏鹰请了个保姆，这样子在一个时间点里，每个小雏鹰都可以同时吃到食物，而且互相不干扰。

回到线程池，控制线程池的并发数可以有效的避免大量的线程池争夺CPU资源而造成堵塞。头脑风暴：还是拿老鹰的例子来讲，妈妈只有一个，要这么一个个喂下去，一些饿坏的小雏鹰等不下去了就要破坏规则，抢在靠前喂食的雏鹰面前，而前面的雏鹰也不是吃软饭的，于是打起来了，场面混乱。老鹰生气了，这么不懂事，谁也别吃了，于是造成了最后谁也没食吃的局面。

3、线程池可以对线程进行管理

线程池可以提供定时、定期、单线程、并发数控制等功能。比如通过ScheduledThreadPool线程池来执行S秒后，每隔N秒执行一次的任务。

5.2线程池的详解

想必看完上面那篇博客，大家可谓赞不绝口，不过可能有些小伙伴还是记不下来，还有些小伙伴觉得好恶心呀，怎么都是厕所啥的呀！哈哈别着急，我来给大家一种好记的办法。

先来讲讲参数最多的那个构造方法，主要是对那几个烦人的参数进行分析。

5.2.1 ThreadPoolExecutor

```
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable>
                           workQueue,
                           ThreadFactory
                           threadFactory,
                           RejectedExecutionHandler
                           handler)
```

这里是7个参数(我们在开发中用的更多的是5个参数的构造方法), OK, 那我们来看看这里七个参数的含义:

corePoolSize 线程池中核心线程的数量

maximumPoolSize 线程池中最大线程数量

keepAliveTime 非核心线程的超时时长, 当系统中非核心线程闲置时间超过keepAliveTime之后, 则会被回收。如果ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true, 则该参数也表示核心线程的超时时长

unit 第三个参数的单位, 有纳秒、微秒、毫秒、秒、分、时、天等

workQueue 线程池中的任务队列, 该队列主要用来存储已经被提交但是尚未执行的任务。存储在这里的任务是由ThreadPoolExecutor的execute方法提交来的。

threadFactory 为线程池提供创建新线程的功能, 这个我们一般使用默认即可

handler 拒绝策略, 当线程无法执行新任务时 (一般是由于线程池中的线程数量已经达到最大数或者线程池关闭导致的), 默认情况下, 当线程池无法处理新线程时, 会抛出一个RejectedExecutionException。

emmmmm....看到那么多烦人的概念, 是不是有点头大了, 我反正是头大了。

这7个参数中, 平常最多用到的是**corePoolSize**、**maximumPoolSize**、**keepAliveTime**、**unit**、**workQueue**. 在这里我主抽出**corePoolSize**、**maximumPoolSize**和**workQueue**三个参数进行详解。

$\text{maximumPoolSize}(\text{最大线程数}) = \text{corePoolSize}(\text{核心线程数}) + \text{noCorePoolSize}(\text{非核心线程数})$;

(1) 当currentSize<corePoolSize时, 没什么好说的, 直接启动一个核心线程并执行任务。

(2) 当`currentSize` \geq `corePoolSize`、并且`workQueue`未满时，添加进来的任务会被安排到`workQueue`中等待执行。

(3) 当`workQueue`已满，但是`currentSize` $<$ `maximumPoolSize`时，会立即开

启一个非核心线程来执行任务。

(4) 当`currentSize` \geq `corePoolSize`、`workQueue`已满、并且`currentSize` $>$ `maximumPoolSize`时，调用`handler`默认抛出`RejectedExecutionException`异常。

什么`currentSize`,`corePoolSize`,`maximumPoolSize`,`workQueue`比来比去的都比迷糊了，哈哈，那我举个烧烤店的例子来想必大家理解起来更快。

夏天了，很热，所以很多烧烤店都会在外面也布置座位，分为室内、室外两个地方可以吃烧烤。（室内有空调电视，而且室内比室外烧烤更加优惠，而且外面下着瓢泼大雨所以顾客会首先选择室内）

`corePoolSize`(烧烤店室内座位), `currentPoolSize` (目前到烧烤店的顾客数量) , `maximumPoolSize` (烧烤店室内+室外+候厅室所有座位) , `workQueue`(烧烤店为顾客专门设置的候厅室)

第(1)种，烧烤店人数不多的时候，室内位置很多，大家都其乐融融，开心的坐在室内吃着烧烤，看着世界杯。

第(2)种，生意不错，室内烧烤店坐无空席，大家都不愿意去外面吃，于是在候厅室里呆着，候厅室位置没坐满。

第(3)种，生意兴隆，室内、候厅室都坐无空席，但是顾客太饿了，剩下的人没办法只好淋着雨吃烧烤，哈哈，好可怜。

第(4)种，生意爆棚，室内、室外、候厅室都坐无空席，在有顾客过来直接赶走。

哈哈是不是很形象。

5.2.2其他线程池的记法

剩下的那四种主要的线程池大概思路，用法在我推荐的博客里都有详细解说，在这里我就不一一道来了，在这里主要是跟大家分享一种特别容易记住这四种线程池的方法，在大家写代码，面试时可以即使想到这四种线程池。

(1) FixedThreadPool:

Fixed中文解释为固定。结合在一起解释固定的线程池，说的更全面点就是，有固定数量线程的线程池。其 corePoolSize=maximumPoolSize，且keepAliveTime为0，适合线程稳定的场所。

(2) SingleThreadPool:

Single中文解释为单一。结合在一起解释单一的线程池，说的更全面点就是，有固定数量线程的线程池，且数量为一，从数学的角度来看 SingleThreadPool应该属于FixedThreadPool的子集。其 corePoolSize=maximumPoolSize=1,且keepAliveTime为0，适合线程同步操作的场所。

(3) CachedThreadPool:

Cached中文解释为储存。结合在一起解释储存的线程池，说的更通俗易懂，既然要储存，其容量肯定是很大，所以他的corePoolSize=0， maximumPoolSize=Integer.MAX_VALUE(2^32-1一个很大的数字)

(4) ScheduledThreadPool:

Scheduled中文解释为计划。结合在一起解释计划的线程池，顾名思义既然涉及到计划，必然涉及到时间。所以ScheduledThreadPool是一个具有定时定期执行任务功能的线程池。

5.3线程池的单例

容我伸个懒腰，该讲本章重点内容了，在此之前，我们对基本语意知识进行了解一下。

什么是单例呢？咳咳。

5.3.1单例

单例模式 (Singleton Pattern) 是 Java 中最简单的设计模式之一。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意事项：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

5.3.2线程池的单例

那么问题来了，我线程池用的好好的，用的时候创建一个，不用就不管他，那为什么要将线程池设计成单例模式呢。那么就要看看你将线程池应用的场所了。一般情况下，整个系统中只需要单种线程池，多个线程公用一个线程池，不会是每创一个线程就要创建一个线程池，那样子你还不如不用线程池呢。

言归正传，咱们来看看如何将线程池设计成单例模式。废话少说上代码

首先在**ThreadPool**类里面实现线程池的创建，我们这里创建的是**FixedThreadPool**线程池（记住构造方法要私有，保证不被其他类实例化）

```
private ThreadPool(int corepoolsize, int maximumpoolsize, long keepalivetime){  
    this.corepoolsize = corepoolsize;  
    this.maximumpoolsize = maximumpoolsize;  
    this.keepalivetime = keepalivetime;  
}  
public void executor(Runnable runnable){  
  
    if (runnable == null){  
        return;  
    }  
}
```

```
    if (mexecutor == null) {
        mexecutor = new
ThreadPoolExecutor(corepoolsize, //核心线程数
                    maximumpoolsize, //最大线程数
                    keepalivetime, //闲置线程存活时间
                    TimeUnit.MILLISECONDS, // 时间单位
                    new LinkedBlockingDeque<Runnable>(),
//线程队列
                    Executors.defaultThreadFactory(), //
线程工厂
                    new ThreadPoolExecutor.AbortPolicy()
//队列已满，而且当前线程数已经超过最大线程数时的异常处理策略
                );
    }
    mexecutor.execute(runnable);
}
```

再然后对ThreadPool内部类，在类里面对他实例化，实现单例

```
// 获取单例的线程池对象
public static ThreadPool getThreadPool() {
    if (mThreadPool == null) {
        synchronized (ThreadManager.class) {
            if (mThreadPool == null) {
                int cpuNum =
Runtime.getRuntime().availableProcessors(); // 获取处理器数量
                int threadNum = cpuNum * 2 + 1; // 根据
cpu数量，计算出合理的线程并发数
                mThreadPool = new
ThreadPool(threadNum, threadNum, 0L);
            }
        }
    }
    return mThreadPool;
}
```

第六节 线程池的优点及其原理

6.1 使用线程池的好处

池化技术应用：线程池、数据库连接池、http连接池等等。

池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。

线程池提供了一种限制、管理资源的策略。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

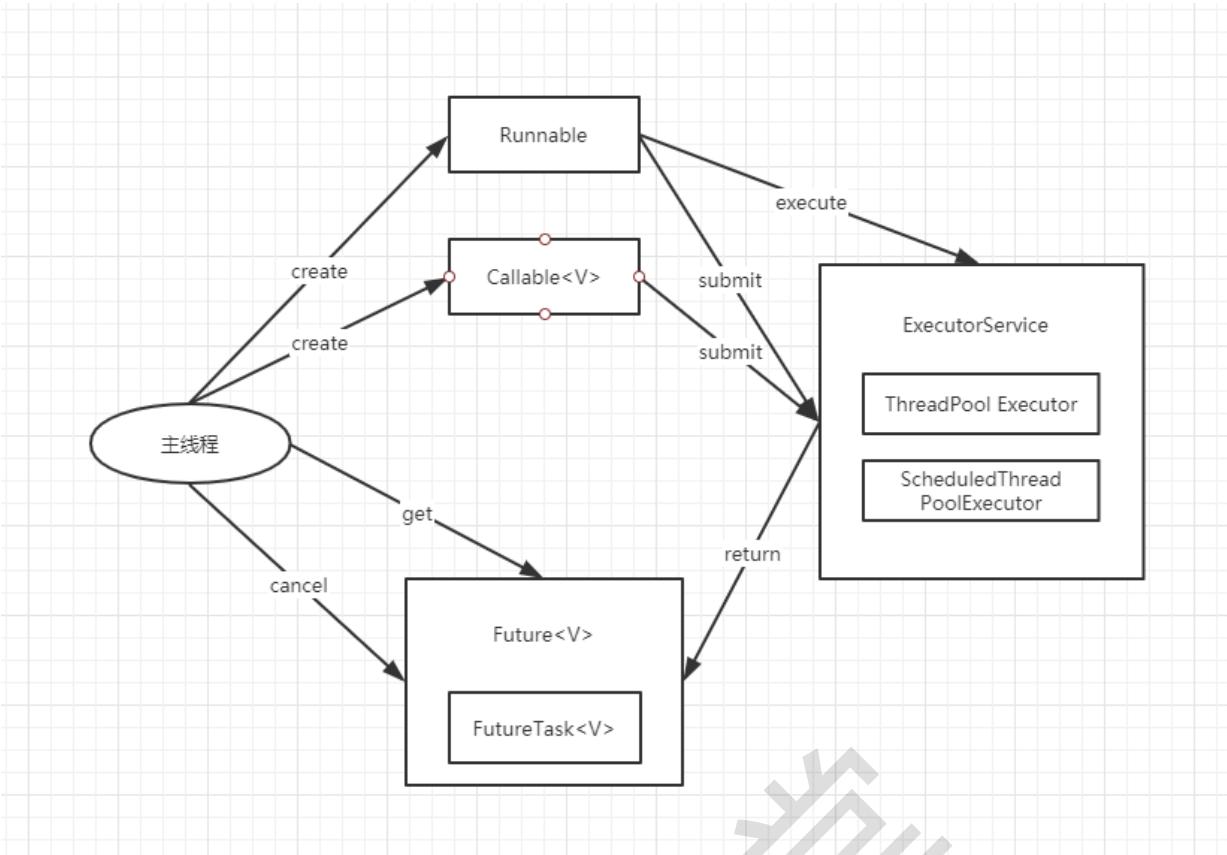
使用线程池的好处：

- **降低资源消耗：**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度：**当任务到达时，可以不需要等待线程创建就能立即执行。
- **提高线程的可管理性：**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，监控和调优。

6.2 Executor框架

Executor框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等，让并发编程变得更加简单。

Executor框架的使用示意图



- **主线程首先要创建实现Runnable或Callable接口的任务对象。**
- **把创建完成的实现Runnable/Callable接口的对象直接交给ExecutorService执行：**

ExecutorService.execute(Runnable command)或者
 ExecutorService.submit(Runnable command)或
 ExecutorService.submit(Callable <T> task).

- **如果执行ExecutorService.submit(...), ExecutorService将返回一个实现Future接口的对象。最后，主线程可以执行FutureTask.get()方法来等待任务执行完成。主线程也可以执行FutureTask.cancel()来取消次任务的执行。**

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
}

```

```
private static final Long KEEP\_ALIVE\_TIME = 1L;

public static void main(String\[ \] args) {
    ThreadPoolExecutor executor \= new
    ThreadPoolExecutor(
        CORE\_POOL\_SIZE,
        MAX\_POOL\_SIZE,
        KEEP\_ALIVE\_TIME,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<>
        (QUEUE\_CAPACITY),
        new
    ThreadPoolExecutor.CallerRunsPolicy());
}

//执行线程代码
executor.shutdown();

}

}
```

CORE_POOL_SIZE:核心线程数定义了最小可以同时运行的线程数量。

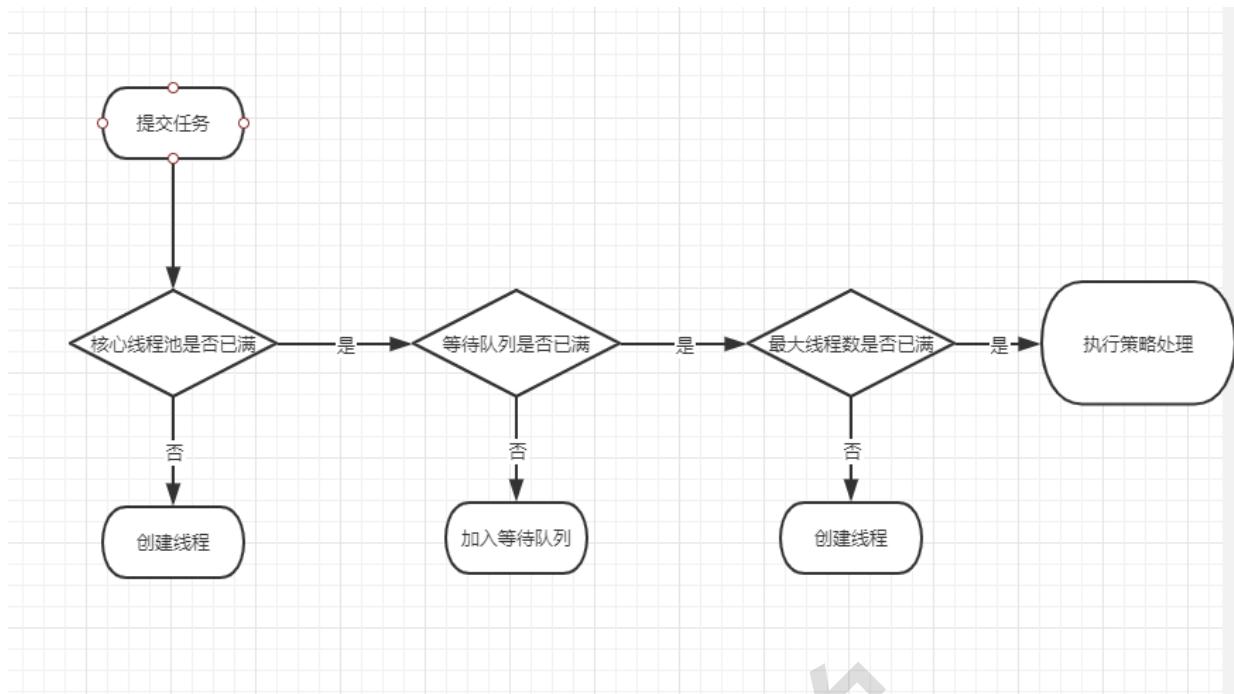
MAX_POOL_SIZE: 当队列中存放的任务到达队列容量的时候，当前可以同时运行的线程数量变为最大线程数。

QUEUE_CAPACITY: 当新任务加入是会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放到队列中。

KEEP_ALIVE_TIME: 当线程池中的线程数量大于核心线程数时，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过**KEEP_ALIVE_TIME**才会被回收销毁。

ThreadPoolExecutor.CallerRunsPolicy(): 调用执行自己的线程运行任务，也就是直接在调用execute方法的线程运行（run）被拒绝的任务，如果执行程序已关闭，则会丢弃任务。因此这种策略会降低新任务的提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果应用程序可以承受此延迟并且不能任务丢弃一个任务请求的话，可以选择这个策略。

线程池分析原理



6.3 线程池大小确定

有一个简单且使用面比较广的公式：

- **CPU密集型任务 (N+1)**：这种任务消耗的主要是CPU资源，可以将线程数设置为 N (CPU核心数) + 1，比CPU核心数多出来一个线程是为了防止线程偶发的缺页中断，或者其他原因导致的任务暂停而带来的影响。一旦任务停止，CPU就会出于空闲状态，而这种情况下多出来一个线程就可以充分利用CPU的空闲时间。
- **I/O密集型 (2N)**：这种任务应用起来，系统大部分时间用来处理I/O交互，而线程在处理I/O的是时间段内不会占用CPU来处理，这时就可以将CPU交给其他线程使用。因此在I/O密集型任务的应用中，可以配置多一些线程，具体计算方是 $2N$ 。

第七节 线程池的优点（重点）

7.1 线程池的优点

1. 线程是稀缺资源，使用线程池可以减少创建和销毁线程的次数，每个工作线程都可以重复使用。

2.可以根据系统的承受能力，调整线程池中工作线程的数量，防止因为消耗过多内存导致服务器崩溃。

7.2线程池的创建

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable>
workQueue,
                          RejectedExecutionHandler
handler)
```

corePoolSize：线程池核心线程数量

maximumPoolSize:线程池最大线程数量

keepAliveTime：当活跃线程数大于核心线程数时，空闲的多余线程最大存活时间

unit：存活时间的单位

workQueue：存放任务的队列

handler：超出线程范围和队列容量的任务的处理程序

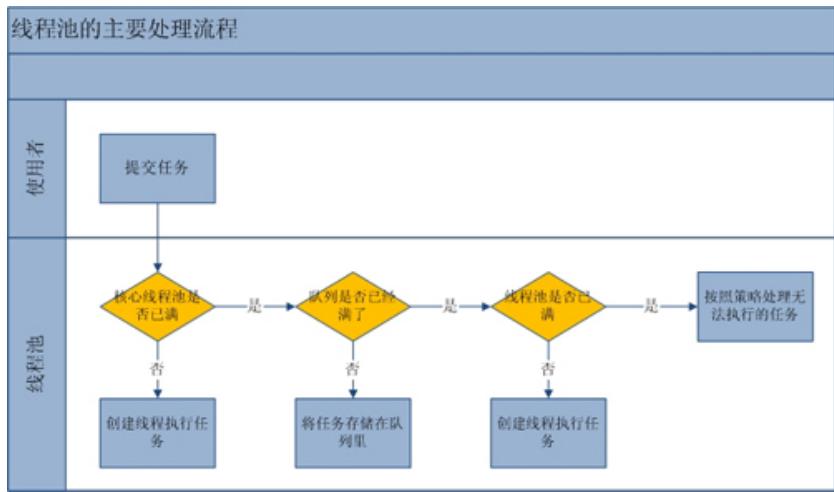
7.3线程池的实现原理

提交一个任务到线程池中，线程池的处理流程如下：

1.判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2.线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3.判断**线程池里的线程**是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。



7.4 线程池的源码解读

7.4.1 ThreadPoolExecutor的execute()方法

```

1 public void execute(Runnable command) {
2     if (command == null)
3         throw new NullPointerException();
4     //如果线程数大于等于基本线程数或者线程创建失败，将任务
加入队列
5     if (poolsize >= corePoolsize ||
!addIfUnderCorePoolSize(command)) {
6         //线程池处于运行状态并且加入队列成功
7         if (runState == RUNNING &&
workQueue.offer(command)) {
8             if (runState != RUNNING || poolsize ==
0)
9                 ensureQueuedTaskHandled(command);
10        }
11        //线程池不处于运行状态或者加入队列失败，则创建线程
(创建的是非核心线程)
12        else if
(!addIfUnderMaximumPoolSize(command))
13            //创建线程失败，则采取阻塞处理的方式
14            reject(command); // is shutdown or
saturated
15        }
16    }
  
```

7.4.2 创建线程的方法：addIfUnderCorePoolSize(command)

```
1 private boolean addIfUnderCorePoolSize(Runnable
firstTask) {
2     Thread t = null;
3     final ReentrantLock mainLock = this.mainLock;
4     mainLock.lock();
5     try {
6         if (poolsize < corePoolSize && runState ==
RUNNING)
7             t = addThread(firstTask);
8     } finally {
9         mainLock.unlock();
10    }
11    if (t == null)
12        return false;
13    t.start();
14    return true;
15 }
```

我们重点来看第7行：

```
1 private Thread addThread(Runnable firstTask) {
2     worker w = new Worker(firstTask);
3     Thread t = threadFactory.newThread(w);
4     if (t != null) {
5         w.thread = t;
6         workers.add(w);
7         int nt = ++poolsize;
8         if (nt > largestPoolsize)
9             largestPoolsize = nt;
10    }
11    return t;
12 }
```

这里将线程封装成工作线程worker，并放入工作线程组里，worker类的方法run方法：

```
public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    } finally {
        workerDone(this);
    }
}
```

worker在执行完任务后，还会通过getTask方法循环获取工作队列里的任务来执行。

我们通过一个程序来观察线程池的工作原理：

1. 创建一个线程

```
1 public class ThreadPoolTest implements Runnable
2 {
3     @Override
4     public void run()
5     {
6         try
7         {
8             Thread.sleep(300);
9         }
10        catch (InterruptedException e)
11        {
12            e.printStackTrace();
13        }
14    }
15 }
```

2. 线程池循环运行16个线程：

```
1 public static void main(String[] args)
2     {
3         LinkedBlockingQueue<Runnable> queue =
4             new LinkedBlockingQueue<Runnable>(5);
5         ThreadPoolExecutor threadPool = new
6         ThreadPoolExecutor(5, 10, 60, TimeUnit.SECONDS, queue);
7         for (int i = 0; i < 16 ; i++)
8         {
9             threadPool.execute(
10                 new Thread(new ThreadPoolTest(),
11                         "Thread".concat(i + "")));
12             System.out.println("线程池中活跃的线程数: " +
13             threadPool.getPoolSize());
14             if (queue.size() > 0)
15             {
16                 System.out.println("-----队
列中阻塞的线程数" + queue.size());
17             }
18         }
19         threadPool.shutdown();
20     }
```

执行结果：

```
线程池中活跃的线程数: 1
线程池中活跃的线程数: 2
线程池中活跃的线程数: 3
线程池中活跃的线程数: 4
线程池中活跃的线程数: 5
线程池中活跃的线程数: 5
-----队列中阻塞的线程数1
线程池中活跃的线程数: 5
-----队列中阻塞的线程数2
线程池中活跃的线程数: 5
-----队列中阻塞的线程数3
线程池中活跃的线程数: 5
-----队列中阻塞的线程数4
线程池中活跃的线程数: 5
```

```
-----队列中阻塞的线程数5  
线程池中活跃的线程数: 6  
-----队列中阻塞的线程数5  
线程池中活跃的线程数: 7  
-----队列中阻塞的线程数5  
线程池中活跃的线程数: 8  
-----队列中阻塞的线程数5  
线程池中活跃的线程数: 9  
-----队列中阻塞的线程数5  
线程池中活跃的线程数: 10  
-----队列中阻塞的线程数5  
Exception in thread "main"  
java.util.concurrent.RejectedExecutionException: Task  
Thread[Thread15,5,main] rejected from  
java.util.concurrent.ThreadPoolExecutor@232204a1[Running,  
pool size = 10, active threads = 10, queued tasks = 5,  
completed tasks = 0]  
at  
java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejec  
tedExecution(ThreadPoolExecutor.java:2047)  
at  
java.util.concurrent.ThreadPoolExecutor.reject(ThreadPool  
Executor.java:823)  
at  
java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoo  
lExecutor.java:1369)  
at test.ThreadTest.main(ThreadTest.java:17)
```

从结果可以观察出：

1. 创建的线程池具体配置为：核心线程数量为5个；全部线程数量为10个；工作队列的长度为5。
2. 我们通过queue.size()的方法来获取工作队列中的任务数。
3. 运行原理：

刚开始都是在创建新的线程，达到核心线程数量5个后，新的任务进来后不再创建新的线程，而是将任务加入工作队列，任务队列到达上线5个后，新的任务又会创建新的普通线程，直到达到线程池最大的线程数量10个，后面的任务则根据配置的饱和策略来处理。我们这里没有具体配置，使用的是默认的配置AbortPolicy:直接抛出异常。

当然，为了达到我需要的效果，上述线程处理的任务都是利用休眠导致线程没有释放！！！

RejectedExecutionHandler：饱和策略

当队列和线程池都满了，说明线程池处于饱和状态，那么必须对新提交的任务采用一种特殊的策略来进行处理。这个策略默认配置是AbortPolicy，表示无法处理新的任务而抛出异常。JAVA提供了4中策略：

1. AbortPolicy：直接抛出异常
2. CallerRunsPolicy：只用调用所在的线程运行任务
3. DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
4. DiscardPolicy：不处理，丢弃掉。

我们现在用第四种策略来处理上面的程序：

```
1 public static void main(String[] args)
2 {
3     LinkedBlockingQueue<Runnable> queue =
4         new LinkedBlockingQueue<Runnable>(3);
5     RejectedExecutionHandler handler = new
6     ThreadPoolExecutor.DiscardPolicy();
7     ThreadPoolExecutor threadPool = new
8     ThreadPoolExecutor(2, 5, 60, TimeUnit.SECONDS,
9     queue, handler);
10    for (int i = 0; i < 9 ; i++)
11    {
12        threadPool.execute(
```

```
11             new Thread(new ThreadPoolTest(),  
12                     "Thread".concat(i + ""));  
13             System.out.println("线程池中活跃的线程数: " +  
14             threadPool.getPoolSize());  
15             if (queue.size() > 0)  
16             {  
17                 System.out.println("-----队  
列中阻塞的线程数" + queue.size());  
18             }  
19         }
```

执行结果：

```
线程池中活跃的线程数: 1  
线程池中活跃的线程数: 2  
线程池中活跃的线程数: 2  
-----队列中阻塞的线程数1  
线程池中活跃的线程数: 2  
-----队列中阻塞的线程数2  
线程池中活跃的线程数: 2  
-----队列中阻塞的线程数3  
线程池中活跃的线程数: 3  
-----队列中阻塞的线程数3  
线程池中活跃的线程数: 4  
-----队列中阻塞的线程数3  
线程池中活跃的线程数: 5  
-----队列中阻塞的线程数3  
线程池中活跃的线程数: 5  
-----队列中阻塞的线程数3
```



这里采用了丢弃策略后，就没有再抛出异常，而是直接丢弃。在某些重要的场景下，可以采用记录日志或者存储到数据库中，而不应该直接丢弃。

设置策略有两种方式：

1.

```
RejectedExecutionHandler handler = new  
ThreadPoolExecutor.DiscardPolicy();  
ThreadPoolExecutor threadPool = new  
ThreadPoolExecutor(2, 5, 60, TimeUnit.SECONDS,  
queue, handler);
```

2.

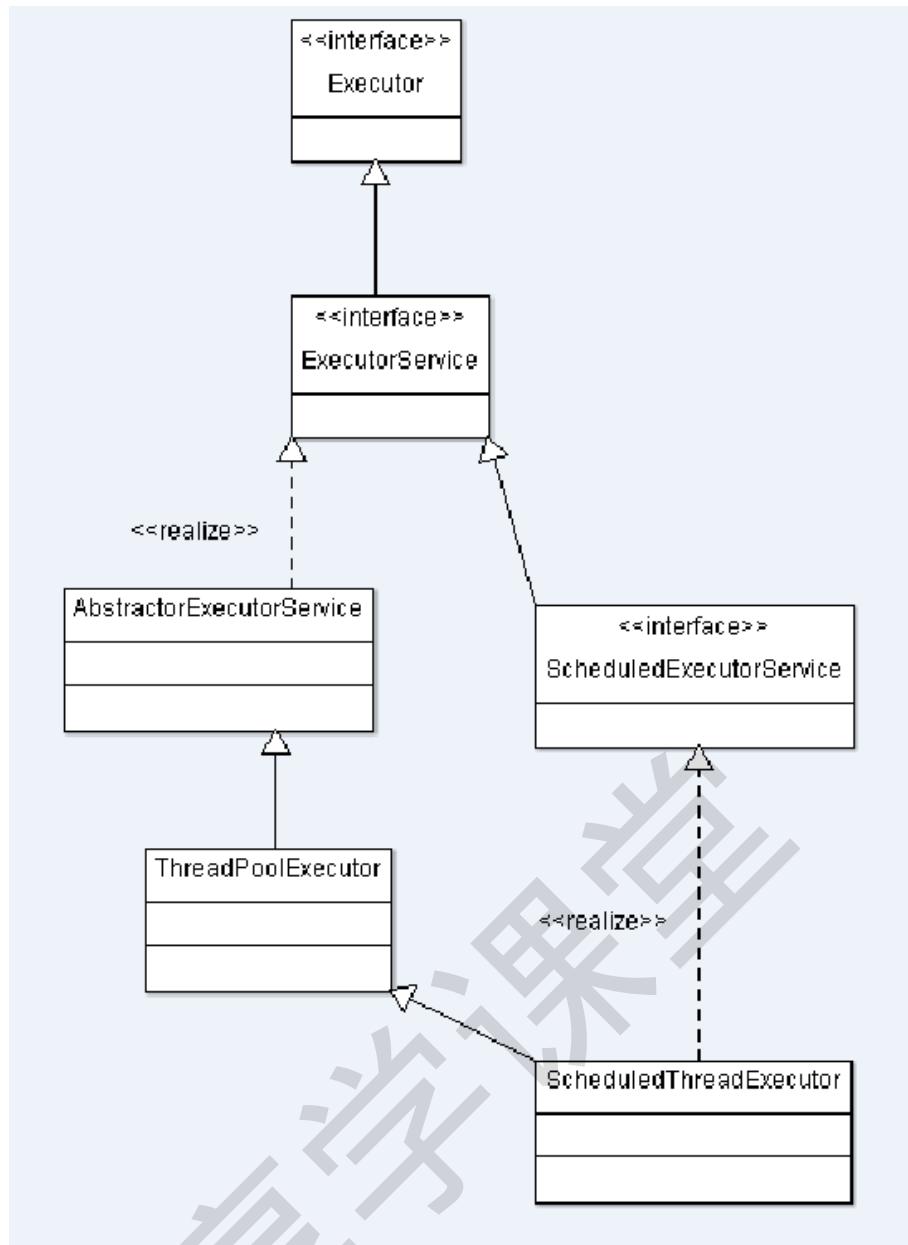
```
ThreadPoolExecutor threadPool = new  
ThreadPoolExecutor(2, 5, 60, TimeUnit.SECONDS, queue);  
threadPool.setRejectedExecutionHandler(new  
ThreadPoolExecutor.AbortPolicy());
```

Executor框架的两级调度模型

在HotSpot VM的模型中，JAVA线程被一对一映射为本地操作系统线程。JAVA线程启动时会创建一个本地操作系统线程，当JAVA线程终止时，对应的操作系统线程也被销毁回收，而操作系统会调度所有线程并将它们分配给可用的CPU。

在上层，JAVA程序会将应用分解为多个任务，然后使用应用级的调度器（Executor）将这些任务映射成固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。

Executor框架类图



在前面介绍的JAVA线程既是工作单元，也是执行机制。而在Executor框架中，我们将工作单元与执行机制分离开来。Runnable和Callable是工作单元（也就是俗称的任务），而执行机制由Executor来提供。这样一来Executor是基于生产者消费者模式的，提交任务的操作相当于生成者，执行任务的线程相当于消费者。

1.从类图上看，Executor接口是异步任务执行框架的基础，该框架能够支持多种不同类型的任务执行策略。

```

public interface Executor {

    void execute(Runnable command);

}
  
```

Executor接口就提供了一个执行方法，任务是Runnable类型，不支持Callable类型。

2.ExecutorService接口实现了Executor接口，主要提供了关闭线程池和submit方法：

```
public interface ExecutorService extends Executor {  
  
    List<Runnable> shutdownNow();  
  
    boolean isTerminated();  
  
    <T> Future<T> submit(Callable<T> task);  
}
```

另外该接口有两个重要的实现类：ThreadPoolExecutor与ScheduledThreadPoolExecutor。

其中ThreadPoolExecutor是线程池的核心实现类，用来执行被提交的任务；而ScheduledThreadPoolExecutor是一个实现类，可以在给定的延迟后运行任务，或者定期执行命令。

在上一篇文章中，我是使用ThreadPoolExecutor来通过给定不同的参数从而创建自己所需的线程池，但是在后面的工作中不建议这种方式，推荐使用Executors工厂方法来创建线程池

这里先来区别线程池和线程组（ThreadGroup与ThreadPoolExecutor）这两个概念：

a、线程组就表示一个线程的集合。

b、线程池是为线程的生命周期开销问题和资源不足问题提供解决方案，主要是用来管理线程。

Executors可以创建3种类型的ThreadPoolExecutor：

SingleThreadExecutor、FixedThreadPool和CachedThreadPool

a、SingleThreadExecutor：单线程线程池

```
ExecutorService threadPool =  
Executors.newSingleThreadExecutor();
```

```
public static ExecutorService newSingleThreadExecutor() {  
    return new FinalizableDelegatedExecutorService  
        (new ThreadPoolExecutor(1, 1,  
            0L,  
            TimeUnit.MILLISECONDS,  
            new  
            LinkedBlockingQueue<Runnable>()));  
}
```

我们从源码来看可以知道，单线程线程池的创建也是通过 ThreadPoolExecutor，里面的核心线程数和线程数都是1，并且工作队列使用的是无界队列。由于是单线程工作，每次只能处理一个任务，所以后面所有的任务都被阻塞在工作队列中，只能一个个任务执行。

b、FixedThreadExecutor：固定大小线程池

```
ExecutorService threadPool =  
Executors.newFixedThreadPool(5);
```

```
public static ExecutorService newFixedThreadPool(int  
nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        0L,  
        TimeUnit.MILLISECONDS,  
        new  
        LinkedBlockingQueue<Runnable>());  
}
```

这个与单线程类似，只是创建了固定大小的线程数量。

c、CachedThreadPool：无界线程池

```
ExecutorService threadPool =  
Executors.newCachedThreadPool();
```

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(  
        Integer.MAX_VALUE,  
        60L,  
        TimeUnit.SECONDS,  
        new  
        SynchronousQueue<Runnable>());  
}
```

无界线程池意味着没有工作队列，任务进来就执行，线程数量不够就创建，与前面两个的区别是：空闲的线程会被回收掉，空闲的时间是60s。这个适用于执行很多短期异步的小程序或者负载较轻的服务器。

Callable、Future、FutureTask详解

Callable与Future是在JAVA的后续版本中引入进来的，Callable类似于Runnable接口，实现Callable接口的类与实现Runnable的类都是可以被线程执行的任务。

二者之间的关系：

Callable是Runnable封装的异步运算任务。

Future用来保存Callable异步运算的结果

FutureTask封装Future的实体类

1.Callable与Runnable的区别

- a、 Callable定义的方法是call，而Runnable定义的方法是run。
- b、 call方法有返回值，而run方法是没有返回值的。
- c、 call方法可以抛出异常，而run方法不能抛出异常。

2.Future

Future表示异步计算的结果，提供了以下方法，主要是判断任务是否完成、中断任务、获取任务执行结果

```
1 public interface Future<V> {  
2  
3     boolean cancel(boolean mayInterruptIfRunning);  
4  
5     boolean isCancelled();  
6  
7     boolean isDone();  
8  
9     V get() throws InterruptedException,  
ExecutionException;  
10  
11    V get(long timeout, TimeUnit unit)  
throws InterruptedException,  
ExecutionException, TimeoutException;  
12  
13 }
```

3.FutureTask

可取消的异步计算，此类提供了对Future的基本实现，仅在计算完成时才能获取结果，如果计算尚未完成，则阻塞get方法。

```
public class FutureTask<V> implements RunnableFuture<V>
```

```
public interface RunnableFuture<V> extends Runnable,  
Future<V>
```

FutureTask不仅实现了Future接口，还实现了Runnable接口，所以不仅可以将FutureTask当成一个任务交给Executor来执行，还可以通过Thread来创建一个线程。

Callable与FutureTask

定义一个callable的任务：

```
1 public class MyCallableTask implements
Callable<Integer>
2 {
3     @Override
4     public Integer call()
5         throws Exception
6     {
7         System.out.println("callable do somothing");
8         Thread.sleep(5000);
9         return new Random().nextInt(100);
10    }
11 }
```

```
1 public class CallableTest
2 {
3     public static void main(String[] args) throws
Exception
4     {
5         Callable<Integer> callable = new
MyCallableTask();
6         FutureTask<Integer> future = new
FutureTask<Integer>(callable);
7         Thread thread = new Thread(future);
8         thread.start();
9         Thread.sleep(100);
10        //尝试取消对此任务的执行
11        future.cancel(true);
12        //判断是否在任务正常完成前取消
13        System.out.println("future is cancel:" +
future.isCancelled());
14        if(!future.isCancelled())
15        {
16            System.out.println("future is cancelled");
17        }
18        //判断任务是否已完成
19        System.out.println("future is done:" +
future.isDone());
20        if(!future.isDone())
```

```
21     {
22         System.out.println("future get=" +
23             future.get());
24     }
25     else
26     {
27         //任务已完成
28         System.out.println("task is done");
29     }
30 }
```

执行结果：

```
callable do somothing
future is cancel:true
future is done:true
task is done
```

这个DEMO主要是通过调用FutureTask的状态设置的方法，演示了状态的变迁。

a、第11行，尝试取消对任务的执行，该方法如果由于任务已完成、已取消则返回false，如果能够取消还未完成的任务，则返回true，该DEMO中由于任务还在休眠状态，所以可以取消成功。

```
future.cancel(true);
```

b、第13行，判断任务取消是否成功：如果在任务正常完成前将其取消，则返回true

```
System.out.println("future is cancel:" +
future.isCancelled());
```

c、第19行，判断任务是否完成：如果任务完成，则返回true，以下几种情况都属于任务完成：正常终止、异常或者取消而完成。

```
**我们的DEMO中，任务是由于取消而导致完成。**
```

```
System.out.println("future is done:" + future.isDone());
```

d、在第22行，获取异步线程执行的结果，我这个DEMO中没有执行到这里，需要注意的是，future.get方法会阻塞当前线程，直到任务执行完成返回结果为止。

```
System.out.println("future get=" + future.get());
```

Callable与Future

```
public class CallableThread implements Callable<String>
{
    @Override
    public String call()
        throws Exception
    {
        System.out.println("进入call方法，开始休眠，休眠时间为：" + System.currentTimeMillis());
        Thread.sleep(10000);
        return "今天停电";
    }

    public static void main(String[] args) throws
Exception
    {
        ExecutorService es =
Executors.newSingleThreadExecutor();
        Callable<String> call = new CallableThread();
        Future<String> fu = es.submit(call);
        es.shutdown();
        Thread.sleep(5000);
        System.out.println("主线程休眠5秒，当前时间" +
System.currentTimeMillis());
        String str = fu.get();
        System.out.println("Future已拿到数据，str=" + str +
";当前时间为：" + System.currentTimeMillis());
    }
}
```

```
    }  
}
```

执行结果：

```
进入call方法，开始休眠，休眠时间为：1478606602676  
主线程休眠5秒，当前时间1478606608676  
Future已拿到数据，str=今天停电；当前时间为：1478606612677
```

这里的future是直接扔到线程池里面去执行的。由于要打印任务的执行结果，所以从执行结果来看，主线程虽然休眠了5s，但是从Call方法执行到拿到任务的结果，这中间的时间差正好是10s，说明get方法会阻塞当前线程直到任务完成。

通过FutureTask也可以达到同样的效果：

```
public static void main(String[] args) throws Exception  
{  
    ExecutorService es =  
    Executors.newSingleThreadExecutor();  
    Callable<String> call = new CallableThread();  
    FutureTask<String> task = new FutureTask<String>  
(call);  
    es.submit(task);  
    es.shutdown();  
    Thread.sleep(5000);  
    System.out.println("主线程等待5秒，当前时间为：" +  
System.currentTimeMillis());  
    String str = task.get();  
    System.out.println("Future已拿到数据，str=" + str +  
";当前时间为：" + System.currentTimeMillis());  
}
```

以上的组合可以给我们带来这样的一些变化：

如有一种场景中，方法A返回一个数据需要10s,A方法后面的代码运行需要20s，但是这20s的执行过程中，只有后面10s依赖于方法A执行的结果。如果与以往一样采用同步的方式，势必会有10s的时间被浪费，如果采用前面两种组合，则效率会提高：

- 1.先把A方法的内容放到Callable实现类的call()方法中
 - 2.在主线程中通过线程池执行A任务
 - 3.执行后面方法中10秒不依赖方法A运行结果的代码
 - 4.获取方法A的运行结果，执行后面方法中10秒依赖方法A运行结果的代码
- 这样代码执行效率一下子就提高了，程序不必卡在A方法处。

第八节 为什么不推荐通过Executors直接创建线程池

阿里发布的 Java开发手册中强制线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式。这是为什么？有4个原因，前2个是主要原因。具体如下：

8.1缓存队列 LinkedBlockingQueue 没有设置固定容量大小

8.1.1Executors.newFixedThreadPool()

创建固定大小的线程池

```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        0L,  
        TimeUnit.MILLISECONDS,  
        new  
    LinkedBlockingQueue<Runnable>());  
}
```

ThreadPoolExecutor 部分参数：

- `corePoolSize`：线程池中核心线程数的最大值。此处为 `nThreads`个。

- maximumPoolSize : 线程池中能拥有最多线程数。此处为 nThreads 个。
- LinkedBlockingQueue 用于缓存任务的阻塞队列。此处没有设置容量大小，默认是 Integer.MAX_VALUE，可以认为是无界的。

问题分析：

从源码中可以看出，虽然表面上 newFixedThreadPool() 中定义了核心线程数和最大线程数都是固定 nThreads 个，但是当线程数量超过 nThreads 时，多余的线程会保存到 LinkedBlockingQueue 中，而 LinkedBlockingQueue 没有无界的，导致其无限增大，最终内存撑爆。

8.1.2 Executors.newSingleThreadExecutor()

创建单个线程池，线程池中只有一个线程。

```
public static ExecutorService newSingleThreadExecutor() {  
    return new FinalizableDelegatedExecutorService  
        (new ThreadPoolExecutor(1, 1,  
                               0L,  
                               TimeUnit.MILLISECONDS,  
                               new  
                               LinkedBlockingQueue<Runnable>()));  
}
```

创建单个线程池，线程池中只有一个线程。

优点： 创建一个单线程的线程池，保证线程的顺序执行；

缺点： 与 newFixedThreadPool() 相同。

总结：

newFixedThreadPool()、newSingleThreadExecutor() **底层代码** 中 LinkedBlockingQueue 没有设置容量大小，默认是 Integer.MAX_VALUE，可以认为是无界的。线程池中多余的线程会被缓存到 LinkedBlockingQueue 中，最终内存撑爆。

8.2最大线程数量是 Integer.MAX_VALUE

8.2.1Executors.newCachedThreadPool()

缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
                                60L, TimeUnit.SECONDS,  
                                new  
SynchronousQueue<Runnable>());  
}
```

ThreadPoolExecutor 部分参数：

- corePoolSize：线程池中核心线程数的最大值。此处为 0 个。
- maximumPoolSize：线程池中能拥有最多线程数。此处为 Integer.MAX_VALUE。可以认为是无限大。

优点：很灵活，弹性的线程池线程管理，用多少线程给多大的线程池，不用后及时回收，用则新建；

缺点：从源码中可以看出，SynchronousQueue() 只能存一个队列，可以认为所有 放到 newCachedThreadPool() 中的线程，不会缓存到队列中，而是直接运行的，由于最大线程数是 Integer.MAX_VALUE，这个数量级可以认为是无限大了，随着执行线程数量的增多 和 线程没有及时结束，最终会将内存撑爆。

8.2.2Executors.newScheduledThreadPool()

创建固定大小的线程，可以延迟或定时的执行任务

```
public static ScheduledExecutorService  
newScheduledThreadPool(  
    int corePoolSize, ThreadFactory threadFactory) {  
    return new ScheduledThreadPoolExecutor(corePoolSize,  
    threadFactory);  
}  
  
// ScheduledThreadPoolExecutor 类的源码:  
public ScheduledThreadPoolExecutor(int corePoolSize,  
    ThreadFactory  
threadFactory) {  
    super(corePoolSize, Integer.MAX_VALUE, 0,  
TimeUnit.NANOSECONDS,  
        new DelayedWorkQueue(), threadFactory);  
}
```

优点：创建一个固定大小线程池，可以定时或周期性的执行任务；

缺点：与 newCachedThreadPool() 相同。

总结：

newCachedThreadPool()、newScheduledThreadPool() 的 **底层代码** 中的最大线程数 (maximumPoolSize) 是 Integer.MAX_VALUE，可以认为是无限大，如果线程池中，执行中的线程没有及时结束，并且不断地有线程加入并执行，最终会将内存撑爆。

8.3 拒绝策略不能自定义（这个不是重点）

它们统一缺点：不支持自定义拒绝策略。

Executors 底层其实是使用的 ThreadPoolExecutor 的方式创建的，但是使用的是 ThreadPoolExecutor 的默认策略，即 AbortPolicy。

```
//默认策略  
private static final RejectedExecutionHandler  
defaultHandler =  
    new AbortPolicy();
```

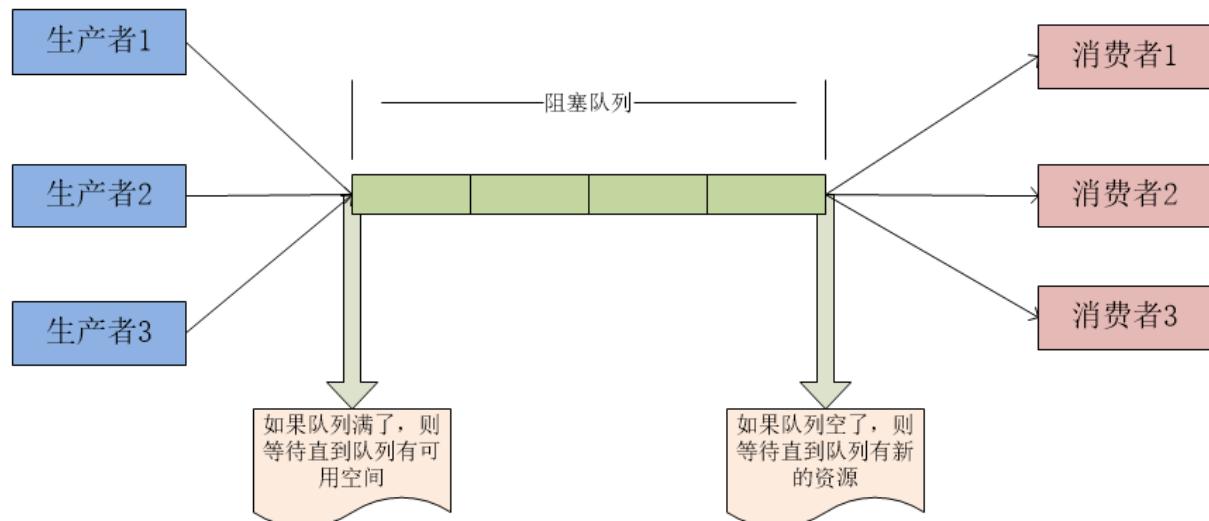
```
//构造函数
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable>
                           workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime,
         unit, workQueue,
         Executors.defaultThreadFactory(),
         defaultHandler);
}
```

8.4 创建线程或线程池时请指定有意义的线程名称，方便出错时回溯（这个不是重点）

第九节 不怕难之BlockingQueue及其实现

9.1 前言

BlockingQueue即阻塞队列，它是基于ReentrantLock，依据它的基本原理，我们可以实现Web中的长连接聊天功能，当然其最常用的还是用于实现生产者与消费者模式，大致如下图所示：



在Java中，`BlockingQueue`是一个接口，它的实现类有`ArrayBlockingQueue`、`DelayQueue`、`LinkedBlockingDeque`、`LinkedBlockingQueue`、`PriorityBlockingQueue`、`SynchronousQueue`等，它们的区别主要体现在存储结构上或对元素操作上的不同，但是对于`take`与`put`操作的原理，却是类似的。

9.2 阻塞与非阻塞

入队

`offer(E e)`: 如果队列没满，立即返回true；如果队列满了，立即返回false-->不阻塞

`put(E e)`: 如果队列满了，一直阻塞，直到队列不满了或者线程被中断-->阻塞

`offer(E e, long timeout, TimeUnit unit)`: 在队尾插入一个元素，如果队列已满，则进入等待，直到出现以下三种情况：-->阻塞

被唤醒

等待时间超时

当前线程被中断

出队

`poll()`: 如果没有元素，直接返回null；如果有元素，出队

`take()`: 如果队列空了，一直阻塞，直到队列不为空或者线程被中断-->阻塞

`poll(long timeout, TimeUnit unit)`: 如果队列不空，出队；如果队列已空且已经超时，返回null；如果队列已空且时间未超时，则进入等待，直到出现以下三种情况：

被唤醒

等待时间超时

当前线程被中断

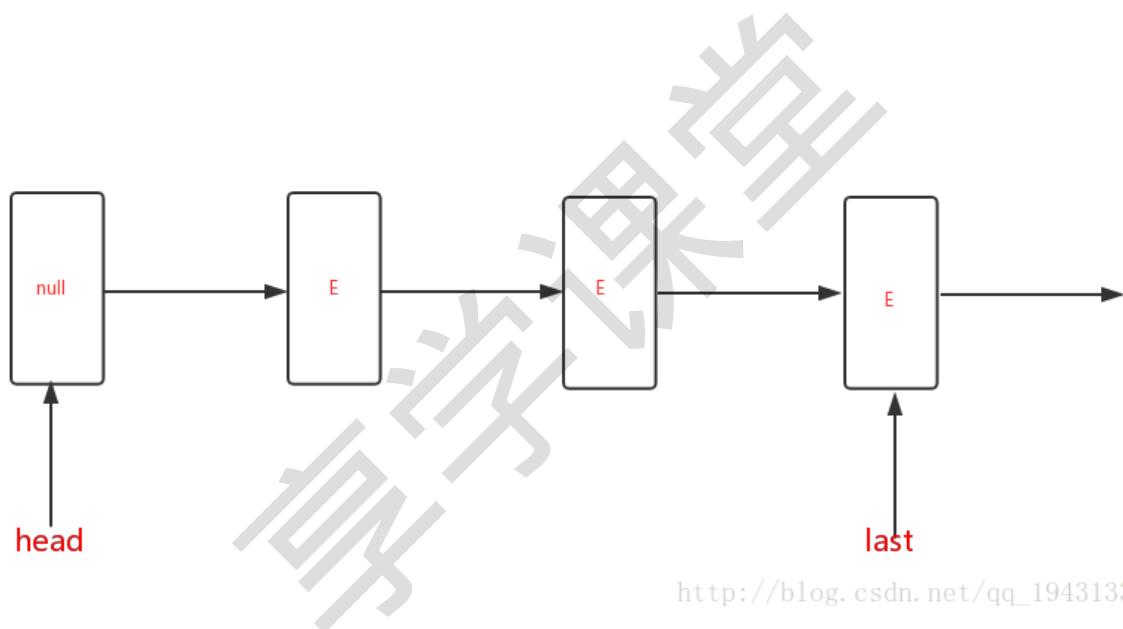
9.3 LinkedBlockingQueue 源码分析

LinkedBlockingQueue是一个基于链表实现的可选容量的阻塞队列。队头的元素是插入时间最长的，队尾的元素是最新插入的。新的元素将会被插入到队列的尾部。

LinkedBlockingQueue的容量限制是可选的，如果在初始化时没有指定容量，那么默认使用int的最大值作为队列容量。

底层数据结构

LinkedBlockingQueue内部是使用链表实现一个队列的，但是却有别于一般的队列，在于该队列至少有一个节点，头节点不含有元素。结构图如下：



原理

LinkedBlockingQueue中维持两把锁，一把锁用于入队，一把锁用于出队，这也就意味着，同一时刻，只能有一个线程执行入队，其余执行入队的线程将会被阻塞；同时，可以有另一个线程执行出队，其余执行出队的线程将会被阻塞。换句话说，虽然入队和出队两个操作同时均只能有一个线程操作，但是可以一个入队线程和一个出队线程共同执行，也就意味着可能同时有两个线程在操作队列，那么为了维持线程安全，

LinkedBlockingQueue使用一个AtomicInteger类型的变量表示当前队列中含有的元素个数，所以可以确保两个线程之间操作底层队列是线程安全的。

源码分析

LinkedBlockingQueue可以指定容量，内部维持一个队列，所以有一个头节点head和一个尾节点last，内部维持两把锁，一个用于入队，一个用于出队，还有锁关联的Condition对象。主要对象的定义如下：

```
//容量，如果没有指定，该值为Integer.MAX_VALUE;  
private final int capacity;  
  
//当前队列中的元素  
private final AtomicInteger count =new AtomicInteger();  
  
//队列头节点，始终满足head.item==null  
transient Node head;  
  
//队列的尾节点，始终满足last.next==null  
private transient Node last;  
  
//用于出队的锁  
private final ReentrantLock takeLock =new ReentrantLock();  
  
//当队列为空时，保存执行出队的线程  
private final Condition notEmpty = takeLock.newCondition();  
  
//用于入队的锁  
private final ReentrantLock putLock =new ReentrantLock();  
  
//当队列满时，保存执行入队的线程  
private final Condition notFull = putLock.newCondition();
```

put(E e)方法

put(E e)方法用于将一个元素插入到队列的尾部，其实现如下：

```
public void put(E e)throws InterruptedException {
```

```
//不允许元素为null  
if (e ==null)  
throw new NullPointerException();  
  
int c = -1;  
  
//以当前元素新建一个节点  
Node node =new Node(e);  
  
final ReentrantLock putLock =this.putLock;  
  
final AtomicInteger count =this.count;  
  
//获得入队的锁  
putLock.lockInterruptibly();  
  
try {  
  
    //如果队列已满，那么将该线程加入到Condition的等待队列中  
    while (count.get() == capacity) {  
  
        notFull.await();  
    }  
  
    //将节点入队  
    enqueue(node);  
  
    //得到插入之前队列的元素个数  
    c = count.getAndIncrement();  
  
    //如果还可以插入元素，那么释放等待的入队线程  
    if (c +1 < capacity){  
  
        notFull.signal();  
    }  
}
```

```
        }finally {  
            //解锁  
            putLock.unlock();  
        }  
  
        //通知出队线程队列非空  
        if (c == 0)  
            signalNotEmpty();  
    }  
}
```

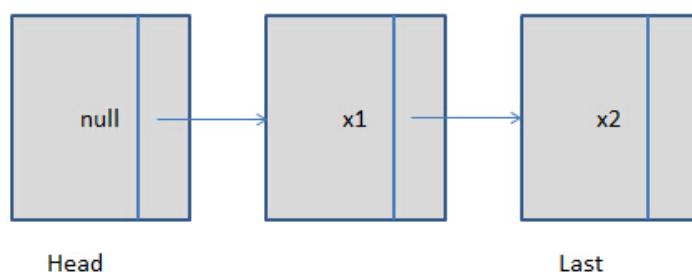
9.3.1 具体入队与出队的原理图：

图中每一个节点前半部分表示封装的数据x，后边的表示指向的下一个引用。

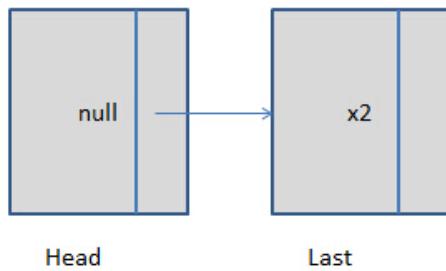


初始化之后，初始化一个数据为null，且head和last节点都是这个节点。

9.3.2 入队两个元素过后



9.3.3出队一个元素后



put方法总结:

1. LinkedBlockingQueue不允许元素为null。
2. 同一时刻，只能有一个线程执行入队操作，因为putLock在将元素插入到队列尾部时加锁了
3. 如果队列满了，那么将会调用notFull的await()方法将该线程加入到Condition等待队列中。await()方法就会释放线程占有的锁，这将导致之前由于被锁阻塞的入队线程将会获取到锁，执行到while循环处，不过可能因为由于队列仍旧是满的，也被加入到条件队列中。
4. 一旦一个出队线程取走了一个元素，并通知了入队等待队列中可以释放线程了，那么第一个加入到Condition队列中的将会被释放，那么该线程将会重新获得put锁，继而执行enqueue()方法，将节点插入到队列的尾部
5. 然后得到插入一个节点之前的元素个数，如果队列中还有空间可以插入，那么就通知notFull条件的等待队列中的线程。
6. 通知出队线程队列为空了，因为插入一个元素之前的个数为0，而插入一个之后队列中的元素就从无变成了有，就可以通知因队列为空而阻塞的出队线程了。

E take()方法

take()方法用于得到队头的元素，在队列为空时会阻塞，知道队列中有元素可取。其实现如下：

```
public E take() throws InterruptedException {
```

```
E x;

int c = -1;

final AtomicInteger count = this.count;

final ReentrantLock takeLock = this.takeLock;

//获取takeLock锁

takeLock.lockInterruptibly();

try {

    • //如果队列为空，那么加入到notEmpty条件的等待队列中

    • while (count.get() == 0) {

        • notEmpty.await();

        • }

    • //得到队头元素

    • x = dequeue();

    • //得到取走一个元素之前队列的元素个数

    • c = count.getAndDecrement();

    • //如果队列中还有数据可取，释放notEmpty条件等待队列中的第一个线程

    • if (c > 1)

    •     notEmpty.signal();
}
```

```
    } finally {  
  
        •     takeLock.unlock();  
  
    }  
  
    //如果队列中的元素从满到非满，通知put线程  
  
    •     if (c == capacity)  
  
    •     signalNotFull();  
  
    return x;  
  
}
```

take方法总结：

当队列为空时，就加入到notEmpty(的条件等待队列中，当队列不为空时就取走一个元素，当取完发现还有元素可取时，再通知一下自己的伙伴（等待在条件队列中的线程）；最后，如果队列从满到非满，通知一下put线程。

remove()方法

remove()方法用于删除队列中一个元素，如果队列中不含有该元素，那么返回false；有的话则删除并返回true。入队和出队都是只获取一个锁，而remove()方法需要同时获得两把锁，其实现如下：

```
public boolean remove(Object o) {  
  
    //因为队列不包含null元素，返回false  
  
    if (o == null) return false;  
  
    //获取两把锁          fullyLock();  
  
    try {
```

```
•      //从头的下一个节点开始遍历

•      for (Node trail = head, p = trail.next;

•          p != null;

•          trail = p, p = p.next) {

•              //如果匹配，那么将节点从队列中移除，trail表示前驱节点

•              if (o.equals(p.item)) {

•                  unlink(p, trail);

•                  return true;
•              }
•          }
•      return false;
}

} finally {

•      //释放两把锁

•      fullyunlock();
}

}
```

```
void fullyLock() {  
  
    putLock.lock();  
  
    takeLock.lock();  
  
}
```

提问：为什么remove()方法同时需要两把锁？

LinkedBlockingQueue总结：

LinkedBlockingQueue是允许两个线程同时在两端进行入队或出队的操作的，但一端同时只能有一个线程进行操作，这是通过两把锁来区分的；

为了维持底部数据的统一，引入了AtomicInteger的一个count变量，表示队列中元素的个数。count只能在两个地方变化，一个是入队的方法（可以+1），另一个是出队的方法（可以-1），而AtomicInteger是原子安全的，所以也就确保了底层队列的数据同步。

9.4ArrayBlockingQueue源码分析

ArrayBlockingQueue底层是使用一个数组实现队列的，并且在构造ArrayBlockingQueue时需要指定容量，也就意味着底层数组一旦创建了，容量就不能改变了，因此ArrayBlockingQueue是一个容量限制的阻塞队列。因此，在队列全满时执行入队将会阻塞，在队列为空时出队同样将会阻塞。

ArrayBlockingQueue的重要字段有如下几个：

```
/** The queued items */  
  
final Object[] items;  
  
/** Main lock guarding all access */  
  
final ReentrantLock lock;  
  
/** Condition for waiting takes */  
  
private final Condition notEmpty;
```

```
/** Condition for waiting puts */
```

```
private final Condition notFull;
```

put(E e)方法

put(E e)方法在队列不满的情况下，将会将元素添加到队列尾部，如果队列已满，将会阻塞，直到队列中有剩余空间可以插入。该方法的实现如下：

```
public void put(E e) throws InterruptedException {  
  
    //检查元素是否为null，如果是，抛出NullPointerException  
  
    checkNotNull(e);  
  
    final ReentrantLock lock = this.lock;  
  
    //加锁  
    lock.lockInterruptibly();  
  
    try {  
  
        •      //如果队列已满，阻塞，等待队列成为不满状态  
  
        •      while (count == items.length)  
  
        •          notFull.await();  
  
        •      //将元素入队  
  
        •      enqueue(e);  
  
    } finally {  
  
        •      lock.unlock();  
  
    }  
}
```

```
}
```

put方法总结:

1. ArrayBlockingQueue不允许元素为null
2. ArrayBlockingQueue在队列已满时将会调用notFull的await()方法释放锁并处于阻塞状态
3. 一旦ArrayBlockingQueue不为满的状态，就将元素入队

E take()方法

take()方法用于取走队头的元素，当队列为空时将会阻塞，直到队列中有元素可取走时将会被释放。其实现如下：

```
public E take() throws InterruptedException {  
  
    final ReentrantLock lock = this.lock;  
  
    //首先加锁  
  
    lock.lockInterruptibly();  
  
    try {  
  
        •      //如果队列为空，阻塞  
  
        •      while (count == 0)  
  
        •      notEmpty.await();  
  
        •      //队列不为空，调用dequeue()出队  
  
        •      return dequeue();  
  
    } finally {  
  
        •      //释放锁  
    }  
}
```

```
    lock.unlock();  
  
}  
  
}
```

take方法总结:

一旦获得了锁之后，如果队列为空，那么将阻塞；否则调用dequeue()出队一个元素。

ArrayBlockingQueue总结:

ArrayBlockingQueue的并发阻塞是通过ReentrantLock和Condition来实现的，ArrayBlockingQueue内部只有一把锁，意味着同一时刻只有一个线程能进行入队或者出队的操作。

9.5总结

在上面分析LinkedBlockingQueue的源码之后，可以与ArrayBlockingQueue做一个比较。

ArrayBlockingQueue:

一个对象数组+一把锁+两个条件

入队与出队都用同一把锁

在只有入队高并发或出队高并发的情况下，因为操作数组，且不需要扩容，性能很高

采用了数组，必须指定大小，即容量有限

LinkedBlockingQueue:

一个单向链表+两把锁+两个条件

两把锁，一把用于入队，一把用于出队，有效的避免了入队与出队时使用一把锁带来的竞争。

在入队与出队都高并发的情况下，性能比ArrayBlockingQueue高很多

采用了链表，最大容量为整数最大值，可看做容量无限

第十节 深入理解ReentrantLock与Condition

锁的概念

从jdk发行1.5版本之后，在原来synchronize的基础上，增加了重入锁ReentrantLock。

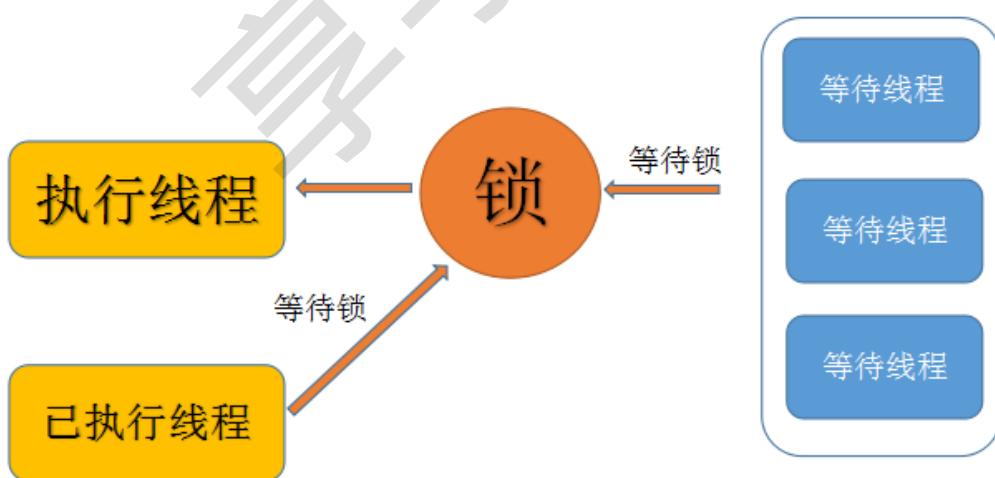
本文就不介绍synchronize了，有兴趣的同学可以去了解一下，本文重点介绍ReentrantLock。

锁是什么？

并发编程的时候，比如说有一个业务是读写操作，那多个线程执行这个业务就会造成已经写入的数据又写一遍，就会造成数据错乱。

所以需要引入锁，进行数据同步，强制使得该业务执行的时候只有一个线程在执行，从而保证不会插入多条重复数据。

一些共享资源也是需要加锁，从而保证数据的一致性。



关于锁的概念，也就不过多篇幅介绍，有很多概念性的东西，需要自己去找本书狠狠啃一啃，本文主要是给大家介绍如何使用锁。

使用ReentrantLock同步

首先来看第一个实例：用两个线程来在控制台有序打出1,2,3。

```

public class FirstReentrantLock {

    public static void main(String[] args) {
        Runnable runnable = new ReentrantLockThread();
        new Thread(runnable, "a").start();
        new Thread(runnable, "b").start();
    }

}

class ReentrantLockThread implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {

System.out.println(Thread.currentThread().getName() + "输出了: " + i);
        }
    }
}

```

执行FirstReentrantLock，查看控制台输出：



```

<terminated> FirstReentr <terminated> FirstReentr
a输出了: 0      a输出了: 0
a输出了: 1      b输出了: 0
b输出了: 0      a输出了: 1
b输出了: 1      b输出了: 1
b输出了: 2      a输出了: 2
a输出了: 2      b输出了: 2

```

可以看到，并没有顺序，杂乱无章。

那使用ReentrantLock加入锁，代码如下：

```

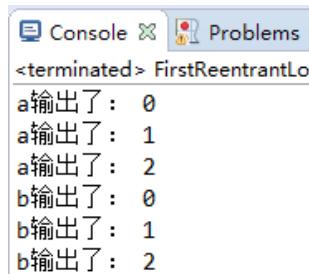
package com.chapter2;

import java.util.concurrent.locks.ReentrantLock;

```

```
/*  
 * @author tangj  
 *  
 *      如何使用ReentrantLock  
 */  
  
public class FirstReentrantLock {  
  
    public static void main(String[] args) {  
        Runnable runnable = new ReentrantLockThread();  
        new Thread(runnable, "a").start();  
        new Thread(runnable, "b").start();  
    }  
  
}  
  
class ReentrantLockThread implements Runnable {  
    // 创建一个ReentrantLock对象  
    ReentrantLock lock = new ReentrantLock();  
  
    @Override  
    public void run() {  
        try {  
            // 使用lock()方法加锁  
            lock.lock();  
            for (int i = 0; i < 3; i++) {  
  
                System.out.println(Thread.currentThread().getName() + "输出了: " + i);  
            }  
        } finally {  
            // 别忘了执行unlock()方法释放锁  
            lock.unlock();  
        }  
    }  
}
```

执行FirstReentrantLock，查看控制台输出：



```
Console Problems
<terminated> FirstReentrantLo
a输出了： 0
a输出了： 1
a输出了： 2
b输出了： 0
b输出了： 1
b输出了： 2
```

有顺序的打印出了0,1,2,0,1,2.

这就是锁的作用，它是互斥的，当一个线程持有锁的时候，其他线程只能等待，待该线程执行结束，再通过竞争得到锁。

使用Condition实现线程等待和唤醒

通常在开发并发程序的时候，会碰到需要停止正在执行业务A，来执行另一个业务B，当业务B执行完成后业务A继续执行。ReentrantLock通过Condition等待/唤醒这样的机制。

相比较synchronize的wait()和notify()/notifyAll()的机制而言，Condition具有更高的灵活性，这个很关键。Condition可以实现多路通知和选择性通知。

当使用notify()/notifyAll()时，JVM会随机通知线程的，具有很大的不可控性，所以建议使用Condition。

Condition使用起来也非常方便，只需要注册到ReentrantLock下面即可。

参考下图：

```
// 实例化一个ReentrantLock对象
private ReentrantLock lock = new ReentrantLock();
// 为线程A注册一个Condition
public Condition conditionA = lock.newCondition();
// 为线程B注册一个Condition
public Condition conditionB = lock.newCondition();
```

接下来，使用Condition来实现等待/唤醒，并且能够唤醒指定线程

先写业务代码：

```
package com.chapter2.howtocondition;
```

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class MyService {

    // 实例化一个ReentrantLock对象
    private ReentrantLock lock = new ReentrantLock();
    // 为线程A注册一个Condition
    public Condition conditionA = lock.newCondition();
    // 为线程B注册一个Condition
    public Condition conditionB = lock.newCondition();

    public void awaitA() {
        try {
            lock.lock();

            System.out.println(Thread.currentThread().getName() + "进入了awaitA方法");
            long timeBefore = System.currentTimeMillis();
            // 执行conditionA等待
            conditionA.await();
            long timeAfter = System.currentTimeMillis();

            System.out.println(Thread.currentThread().getName()+"被唤醒");
            System.out.println(Thread.currentThread().getName() + "等待了: " + (timeAfter - timeBefore)/1000+"s");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void awaitB() {
        try {
            lock.lock();
```

```
System.out.println(Thread.currentThread().getName() + "进入了awaitB方法");
    long timeBefore = System.currentTimeMillis();
    // 执行conditionB等待
    conditionB.await();
    long timeAfter = System.currentTimeMillis();

System.out.println(Thread.currentThread().getName()+"被唤醒");
}

System.out.println(Thread.currentThread().getName() + "等待了: " + (timeAfter - timeBefore)/1000+"s");
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    lock.unlock();
}
}

public void signalAllA() {
try {
    lock.lock();
    System.out.println("启动唤醒程序");
    // 唤醒所有注册conditionA的线程
    conditionA.signalAll();
} finally {
    lock.unlock();
}
}

public void signalAllB() {
try {
    lock.lock();
    System.out.println("启动唤醒程序");
    // 唤醒所有注册conditionB的线程
    conditionB.signalAll();
} finally {
```

```
        lock.unlock();
    }
}

}
```

分别实例化了两个Condition对象，都是使用同一个lock注册。注意conditionA对象的等待和唤醒只对使用了conditionA的线程有用，同理conditionB对象的等待和唤醒只对使用了conditionB的线程有用。

继续写两个线程的代码：

```
package com.chapter2.howtocondition;

public class MyServiceThread1 implements Runnable {

    private MyService service;

    public MyServiceThread1(MyService service) {
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitA();
    }

}
```

注意：MyServiceThread1 使用了awaitA()方法，持有的是conditionA！

```
package com.chapter2.howtocondition;

public class MyServiceThread2 implements Runnable {

    private MyService service;

    public MyServiceThread2(MyService service) {
        this.service = service;
    }

}
```

```
    @Override  
    public void run() {  
        service.awaitB();  
    }  
}
```

注意：MyServiceThread2 使用了awaitB()方法，持有的是conditionB！

最后看启动类：

```
package com.chapter2.howtocondition;  
  
public class ApplicationCondition {  
  
    public static void main(String[] args) throws  
InterruptedException {  
    MyService service = new MyService();  
    Runnable runnable1 = new  
MyServiceThread1(service);  
    Runnable runnable2 = new  
MyServiceThread2(service);  
  
    new Thread(runnable1, "a").start();  
    new Thread(runnable2, "b").start();  
  
    // 线程sleep2秒钟  
    Thread.sleep(2000);  
    // 唤醒所有持有conditionA的线程  
    service.signal1A();  
  
    Thread.sleep(2000);  
    // 唤醒所有持有conditionB的线程  
    service.signal1B();  
}  
}
```

执行ApplicationCondition ,来看控制台输出结果：

```
Console Problems
<terminated> ApplicationCondi
a进入了awaitA方法
b进入了awaitB方法
启动唤醒程序
a被唤醒
a等待了: 2s
启动唤醒程序
b被唤醒
b等待了: 4s
```

a和b都进入各自的await()方法。首先执行的是

```
Thread.sleep(2000);
// 唤醒所有持有conditionA的线程
service.signalA();
```

使用conditionA的线程被唤醒，而后再唤醒使用conditionB的线程。

学会使用Condition,那来用它实现生产者消费者模式

生产者和消费者

首先来看业务类的实现：

```
package com.chapter2.consumeone;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Service {

    private Lock lock = new ReentrantLock();
    private boolean flag = false;
    private Condition condition = lock.newCondition();
    // 以此为衡量标志
    private int number = 1;

    /**
     * 生产者生产
     */
    public void produce() {
```

```
try {
    lock.lock();
    while (flag == true) {
        condition.await();
    }

System.out.println(Thread.currentThread().getName() + "----生产-----");
    number++;
    System.out.println("number: " + number);
    System.out.println();
    flag = true;
    // 提醒消费者消费
    condition.signalAll();
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    lock.unlock();
}

}

/*
 * 消费者消费生产的物品
 */
public void consume() {
try {
    lock.lock();
    while (flag == false) {
        condition.await();
    }

System.out.println(Thread.currentThread().getName() + "----消费-----");
    number--;
    System.out.println("number: " + number);
    System.out.println();
    flag = false;
    // 提醒生产者生产
}
```

```
        condition.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

生产者线程代码：

```
package com.chapter2.consumeone;

/*
 * 生产者线程
 *
 * @author tangj
 */
public class MyThreadProduce implements Runnable {

    private Service service;

    public MyThreadProduce(Service service) {
        this.service = service;
    }

    @Override
    public void run() {
        for (;;) {
            service.produce();
        }
    }
}
```

消费者线程代码：

```
package com.chapter2.consumeone;

/*
 * 消费者线程
 *
 * @author tangj
 *
 */
public class MyThreadConsume implements Runnable {

    private Service service;

    public MyThreadConsume(Service service) {
        super();
        this.service = service;
    }

    @Override
    public void run() {
        for (;;) {
            service.consume();
        }
    }
}
```

启动类：

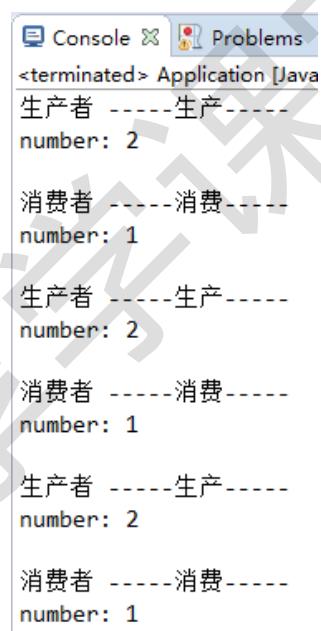
```
package com.chapter2.consumeone;

public class Application {

    public static void main(String[] args) {
        Service service = new Service();
        Runnable produce = new MyThreadProduce(service);
        Runnable consume = new MyThreadConsume(service);
        new Thread(produce, "生产者").start();
        new Thread(consume, "消费者").start();
    }

}
```

执行Application,看控制台的输出：



```
Console Problems
<terminated> Application [Java]
生产者 -----生产-----
number: 2

消费者 -----消费-----
number: 1

生产者 -----生产-----
number: 2

消费者 -----消费-----
number: 1

生产者 -----生产-----
number: 2

消费者 -----消费-----
number: 1
```

因为采用了无限循环，生产者线程和消费者线程会一直处于工作状态，可以看到，生产者线程执行完毕后，消费者线程就会执行，以这样的交替顺序，

而且的number也遵循者生产者生产+1，消费者消费-1的一个状态。这个就是使用ReentrantLock和Condition来实现的生产者消费者模式。

顺序执行线程

充分发掘Condition的灵活性，可以用它来实现顺序执行线程。

来看业务类代码：

```
package com.chapter2.sequencethread;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class Service {

    // 通过nextThread控制下一个执行的线程
    private static int nextThread = 1;
    private ReentrantLock lock = new ReentrantLock();
    // 有三个线程，所有注册三个Condition
    Condition conditionA = lock.newCondition();
    Condition conditionB = lock.newCondition();
    Condition conditionC = lock.newCondition();

    public void excuteA() {
        try {
            lock.lock();
            while (nextThread != 1) {
                conditionA.await();
            }
            System.out.println(Thread.currentThread().getName() + " 工
作");
            nextThread = 2;
            conditionB.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void excuteB() {
        try {
            lock.lock();
        }
```

```
        while (nextThread != 2) {
            conditionB.await();
        }

System.out.println(Thread.currentThread().getName() + " 工
作");
        nextThread = 3;
        conditionC.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void excuteC() {
    try {
        lock.lock();
        while (nextThread != 3) {
            conditionC.await();
        }
    }

System.out.println(Thread.currentThread().getName() + " 工
作");
        nextThread = 1;
        conditionA.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}
```

这里可以看到，注册了三个Condition，分别用于三个线程的等待和通知。

启动类代码：

```
package com.chapter2.sequencethread;

/*
 * 线程按顺序执行
 *
 * @author tangj
 *
 */
public class Application {

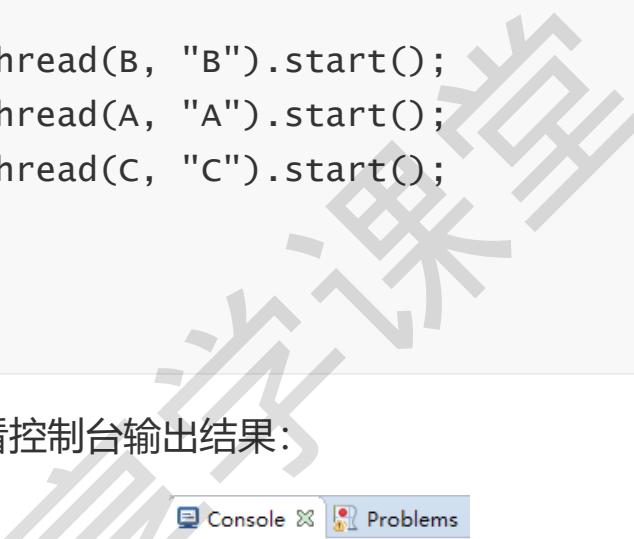
    private static Runnable getThreadA(final Service
service) {
        return new Runnable() {
            @Override
            public void run() {
                for (;;) {
                    service.executeA();
                }
            }
        };
    }

    private static Runnable getThreadB(final Service
service) {
        return new Runnable() {
            @Override
            public void run() {
                for (;;) {
                    service.executeB();
                }
            }
        };
    }

    private static Runnable getThreadC(final Service
service) {
        return new Runnable() {
            @Override
```

```
public void run() {  
    for (;;) {  
        service.execute();  
    }  
}  
};  
  
public static void main(String[] args) {  
    Service service = new Service();  
    Runnable A = getThreadA(service);  
    Runnable B = getThreadB(service);  
    Runnable C = getThreadC(service);  
  
    new Thread(B, "B").start();  
    new Thread(A, "A").start();  
    new Thread(C, "C").start();  
}  
}
```

运行启动类，查看控制台输出结果：



```
Console Problems  
<terminated> Application (1)  
A 工作  
B 工作  
C 工作  
A 工作
```

A,B,C三个线程一直按照顺序执行。

总结

学会使用锁是学好多线程的基础，ReentrantLock相比较关键字synchronize而言，更加而且可控，所以还是推荐大家使用ReentrantLock。

第十一节 Java多线程：线程间通信之Lock

Java5之后，Java在内置关键字synchronized的基础上又增加了一个新的处理锁的方式，Lock类。

由于在[Java线程间通信：volatile与synchronized](#)中，我们已经详细的了解了synchronized，所以我们现在主要介绍一下Lock，以及将Lock与synchronized进行一下对比。

11.1 synchronized的缺陷

synchronized修饰的代码只有获取锁的线程才能够执行，其他线程只能等待该线程释放锁。一个线程释放锁的情况有以下方式：

- 获取锁的线程完成了synchronized修饰的代码块的执行。
- 线程执行时发生异常，JVM自动释放锁。

我们在[Java多线程的生命周期，实现与调度](#)中谈过，锁会因为等待I/O，sleep()方法等原因被阻塞而不释放锁，此时如果线程还处于用synchronized修饰的代码区域里，那么其他线程只能等待，这样就影响了效率。因此Java提供了Lock来实现另一个机制，即不让线程无限期的等待下去。

思考一个情景，当多线程读写文件时，读操作和写操作会发生冲突，写操作和写操作会发生冲突，但读操作和读操作不会有冲突。如果使用synchronized来修饰的话，就很可能造成多个读操作无法同时进行的可能（如果只用synchronized修饰写方法，那么可能造成读写冲突，如果同时修饰了读写方法，则会有读读干扰）。此时就需要用到Lock，换言之Lock比synchronized提供了更多的功能。

使用Lock需要注意以下两点：

- Lock不是语言内置的，synchronized是Java关键字，为内置特性，Lock是一个类，通过这个类可以实现同步访问。
- 采用synchronized时我们不需要手动去控制加锁和释放，系统会自动控制。而使用Lock类，我们需要手动的加锁和释放，不主动释放可能会造成死锁。实际上Lock类的使用某种意义上讲要比synchronized更加直观。

11.2 Lock类接口设计

Lock类本身是一个接口，其方法如下：

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit) throws  
InterruptedException;  
    void unlock();  
    Condition newCondition();  
}
```

下面依次讲解一下其中各个方法。

- **lock()** 方法使用最多，作用是用于获取锁，如果锁已经被其他线程获得，则等待。

通常情况下，lock使用以下方式去获取锁：

```
Lock lock = ...;  
lock.lock();  
try{  
    //处理任务  
}catch(Exception ex){  
  
}finally{  
    lock.unlock(); //释放锁  
}
```

- **lockInterruptibly()** 和**lock()**的区别是lockInterruptibly()锁定的线程处于等待状态时，允许线程的打断操作，线程使用Thread.interrupt()打断该线程后会直接返回并抛出一个InterruptedException(); lock()方法锁定对象时如果在等待时检测到线程使用Thread.interrupt()，仍然会继续尝试获取锁，失败则继续休眠，只是在成功获取锁之后在把当前线程置为interrupt状态。也就使说，当两个线程同时通过lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。
因此，lockInterruptibly()方法必须实现catch(InterruptedException e)代码块。常见使用方式如下：

```
public void method() throws InterruptedException {  
    lock.lockInterruptibly();  
    try {  
        //.....  
    }  
    finally {  
        lock.unlock();  
    }  
}
```

- **tryLock()** 和**lock()**最大的不同是具有返回值，或者说，它不去等待锁。如果它成功获取锁，那么返回true；如果它无法成功获取锁，则返回false。

通常情况下，tryLock使用方式如下：

```

Lock lock = ...;
if(lock.tryLock()) {
    try{
        //处理任务
    }catch(Exception ex){
        //如果不能获取锁，则直接做其他事情
    }
}

```

- **tryLock(long time, TimeUnit unit)** 则是介于二者之间，用户设定一个等待时间，如果在这个时间内获取到了锁，则返回true，否则返回false结束。
- **unlock()** 从上面的代码里我们也看到，unlock()一般放在异常处理操作的finally字符控制的代码块中。我们要记得Lock和synchronized的区别，防止产生死锁。
- **newCondition()** 该方法我们放到后面讲。

11.3 ReentrantLock可重入锁

11.3.1 ReentrantLock概述

ReentrantLock译为“可重入锁”，我们在[Java多线程：synchronized的可重入性](#)中已经明白了什么是可重入以及理解了synchronized的可重入性。ReentrantLock是唯一实现Lock接口的类。

11.3.2 ReentrantLock使用

考虑到以下情景，一个仅出售双人票的演唱会进行门票出售，有三个售票口同时进行售票，买票需要100ms时间，每张票出票需要100ms时间。该如何设计这个情景？

```

package com.cielo.LockTest;

import java.util.concurrent.locks.Lock;

```

```
import java.util.concurrent.locks.ReentrantLock;

import static java.lang.Thread.sleep;

/**
 * Created by 63289 on 2017/4/10.
 */
class SoldTicket implements Runnable {
    Lock lock = new ReentrantLock(); // 使用可重入锁
    private volatile Integer ticket; // 保证从主内存获取

    SoldTicket(Integer ticket) {
        this.ticket = ticket; // 提供票数
    }

    private void sold() {
        lock.lock(); // 锁定操作放在try代码块外
        try {
            if (ticket <= 0) return; // 当ticket==2时可能有多个线程进入sold方法，一个线程运行后另外两个线程需要退出。
            sleep(200); // 买票0.1s，出票0.1s
            --ticket;
            System.out.println("The first ticket is sold
by "+Thread.currentThread().getId()+" , "+ticket+" tickets
leave."); // 获取线程id来识别出票站。
            sleep(100); // 出票0.1s
            --ticket;
            System.out.println("The second ticket is sold
by "+Thread.currentThread().getId()+" , "+ticket+" tickets
leave.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    @Override
```

```
public void run() {
    while (ticket > 0) {
        sold();
    }
}

public class LockTest {
    public static void main(String[] args) {
        SoldTicket soldTicket = new SoldTicket(20);
        new Thread(soldTicket).start();
        new Thread(soldTicket).start();
        new Thread(soldTicket).start();
    }
}
```

上面这段代码结果如下：

```
The first ticket is sold by 11, 19 tickets leave.
The second ticket is sold by 11, 18 tickets leave.
The first ticket is sold by 13, 17 tickets leave.
The second ticket is sold by 13, 16 tickets leave.
The first ticket is sold by 13, 15 tickets leave.
The second ticket is sold by 13, 14 tickets leave.
The first ticket is sold by 12, 13 tickets leave.
The second ticket is sold by 12, 12 tickets leave.
The first ticket is sold by 11, 11 tickets leave.
The second ticket is sold by 11, 10 tickets leave.
The first ticket is sold by 11, 9 tickets leave.
The second ticket is sold by 11, 8 tickets leave.
The first ticket is sold by 13, 7 tickets leave.
The second ticket is sold by 13, 6 tickets leave.
The first ticket is sold by 13, 5 tickets leave.
The second ticket is sold by 13, 4 tickets leave.
The first ticket is sold by 13, 3 tickets leave.
The second ticket is sold by 13, 2 tickets leave.
The first ticket is sold by 13, 1 tickets leave.
The second ticket is sold by 13, 0 tickets leave.
```

如果我们不对售票操作进行锁定，则会有以下几个问题：

- 出售第一张票后其他机器出了另一张票，导致票没有成对卖。
- 已经无票后仍有机器出票造成混乱。

显然，本题的情景用synchronized也可以很容易的实现，实际上Lock有别于synchronized的主要点是lockInterruptibly()和tryLock()这两个可以对锁进行控制的方法。

11.4 ReadWriteLock读写锁

11.4.1 ReadWriteLock接口

回到开头synchronized缺陷的介绍，实际上，Lock接口的重要衍生接口ReadWriteLock即是解决这一问题。ReadWriteLock定义很简单，仅有两个接口：

```
public interface ReadwriteLock {  
    /**  
     * Returns the lock used for reading.  
     *  
     * @return the lock used for reading.  
     */  
    Lock readLock();  
  
    /**  
     * Returns the lock used for writing.  
     *  
     * @return the lock used for writing.  
     */  
    Lock writeLock();  
}
```

即是它只提供了readLock()和writeLock()两个操作，这两个操作均返回一个Lock类的实例。两个操作一个获取读锁，一个获取写锁，将读写分开进行操作。ReadWriteLock将读写的锁分开，可以让多个读操作并行，这就大大提高了效率。使用ReadWriteLock时，用读锁去控制读操作，写锁控

制写操作，进而实现了一个可以在如下的大量读少量写且读者优先的情景运行的锁。

11.4.2 ReentrantReadWriteLock可重入读写锁

ReentrantReadWriteLock是ReadWriteLock的唯一实例。同时提供了很多操作方法。ReentrantReadWriteLock接口实现的读锁写锁进入有如下要求：

11.4.2.1 线程进入读锁的要求

- 没有其他线程的写锁。
- 没有锁请求或调用写请求的线程正是该线程。

11.4.2.2 线程进入写锁的要求

- 没有其他线程的读锁。
- 没有其他线程的写锁。

11.4.2.3 读写锁使用示例

```
private SomeClass someClass; //资源
private final ReadwriteLock readwriteLock = new
ReentrantReadWriteLock(); //创建锁
private final Lock readLock = readwriteLock.readLock(); // //读锁
private final Lock writeLock =
readwriteLock.writeLock(); // //写锁
//读方法
readLock.lock();
try {
    result = someClass.someMethod();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    readLock.unlock();
}
//写方法，产生新的SomeClass实例tempSomeClass
writeLock.lock();
```

```
try{
    this.someClass = tempSomeClass;//更新
} catch (Exception e) {
    e.printStackTrace();
} finally{
    writeLock.unlock();
}
```

11.5 公平锁

公平锁即当多个线程等待的一个资源的锁释放时，线程不是随机的获取资源而是等待时间最久的线程获取资源(FIFO)。Java中，`synchronized`是一个非公平锁，无法保证锁的获取顺序。`ReentrantLock`和`ReentrantReadWriteLock`默认也是非公平锁，但可以设置成公平锁。我们前面的实例中初始化`ReentrantLock`和`ReentrantReadWriteLock`时都是无参数的。实际上，它们提供一个默认的boolean变量`fair`，为true则为公平锁，为false则为非公平锁，默認為false。因此，当我们想将其实现为公平锁时，仅需要初始化时赋值true。即：

```
Lock lock = new ReentrantLock(true);
```

考虑前面卖票的实例，如果改为公平锁（尽管这和情景无关），则结果输出非常整齐如下：

```
The first ticket is sold by 11, 19 tickets leave.
The second ticket is sold by 11, 18 tickets leave.
The first ticket is sold by 12, 17 tickets leave.
The second ticket is sold by 12, 16 tickets leave.
The first ticket is sold by 13, 15 tickets leave.
The second ticket is sold by 13, 14 tickets leave.
The first ticket is sold by 11, 13 tickets leave.
The second ticket is sold by 11, 12 tickets leave.
The first ticket is sold by 12, 11 tickets leave.
The second ticket is sold by 12, 10 tickets leave.
The first ticket is sold by 13, 9 tickets leave.
The second ticket is sold by 13, 8 tickets leave.
The first ticket is sold by 11, 7 tickets leave.
The second ticket is sold by 11, 6 tickets leave.
```

```
The first ticket is sold by 12, 5 tickets leave.  
The second ticket is sold by 12, 4 tickets leave.  
The first ticket is sold by 13, 3 tickets leave.  
The second ticket is sold by 13, 2 tickets leave.  
The first ticket is sold by 11, 1 tickets leave.  
The second ticket is sold by 11, 0 tickets leave.
```

11.6 Lock和synchronized的选择

- synchronized是内置语言实现的关键字，Lock是为了实现更高级锁功能而提供的接口。
- Lock实现了tryLock等接口，线程可以不用一直等待。
- synchronized发生异常时自动释放占有的锁，Lock需要在finally块中手动释放锁。因此从安全性角度讲，既可以用Lock又可以用synchronized时(即不需要锁的更高级功能时)使用synchronized更保险。
- Lock可以通过lockInterruptibly()接口实现可中断锁。
- 由于Lock提供了时间限制同步，可被打断同步等机制，线程激烈竞争时Lock的性能远优于synchronized，即有大量线程时推荐使用Lock。在竞争不激烈时，由于synchronized的编译器优化更好，性能更佳。
- ReentrantReadWriteLock实现了封装好的读写锁用于大量读少量写读者优先情景解决了synchronized读写情景难以实现问题。

11.7 参考文章

[Java并发编程：Lock](#)

[lock和lockInterruptibly](#)

[说说ReentrantReadWriteLock](#)

第十二节 Synchronized 关键字原理

12.1前言

Synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外一篇博客[Java：这是一份全面 & 详细的 Synchronized关键字 学习指南](#)从多个方面介绍Synchronized的原理和使用方式等，值得大家阅读。

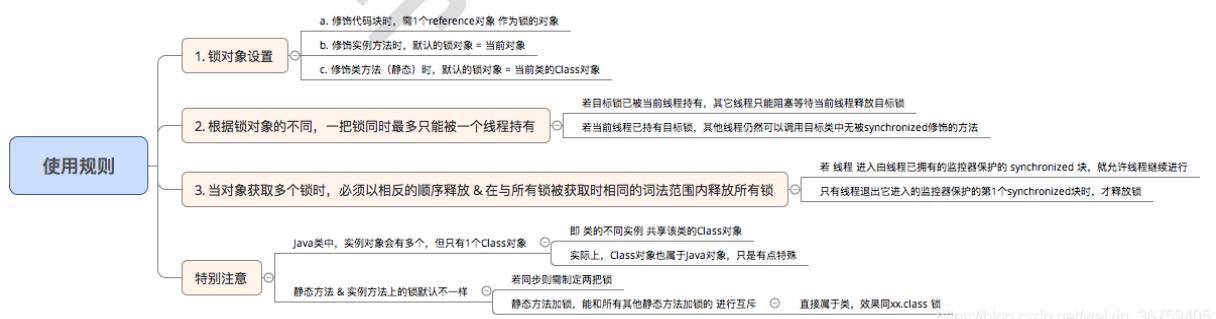
接下来我们正式进入本篇博客的主题。

学习Java的小伙伴都知道synchronized关键字是解决并发问题常用解决方案，常用的有以下三种使用方式：

- 修饰代码块，即同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象。
- 修饰普通方法，即同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象。
- 修饰静态方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象。

关于synchronized的使用方式以及三种锁的区别在[学习指南](#)中讲解的十分清楚。

具体使用规则如下：



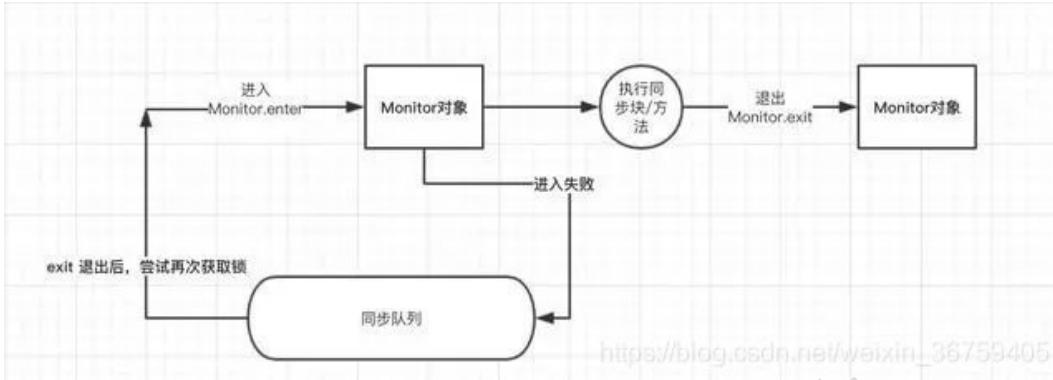
12.2Synchronized 原理

实现原理：JVM 是通过进入、退出 **对象监视器(Monitor)** 来实现对方方法、同步块的同步的，而对象监视器的本质依赖于底层操作系统的 **互斥锁 (Mutex Lock)** 实现。

具体实现是在编译之后在同步方法调用前加入一个`monitor.enter`指令，在退出方法和异常处插入`monitor.exit`的指令。

对于没有获取到锁的线程将会阻塞到方法入口处，直到获取锁的线程`monitor.exit`之后才能尝试继续获取锁。

流程图如下：



通过一段代码来演示：

```
public static void main(String[] args) {  
    synchronized (Synchronize.class){  
        System.out.println("Synchronize");  
    }  
}
```

使用javap -c Synchronize可以查看编译之后的具体信息。

```
liuxpdeMacBook-Pro:classes liuxp$ javap -c com.paddx.test.concurrent.SynchronizedDemo
Compiled from "SynchronizedDemo.java"
public class com.paddx.test.concurrent.SynchronizedDemo {
    public com.paddx.test.concurrent.SynchronizedDemo();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
            4: return

    public void method();
        Code:
            0: aload_0
            1: dup
            2: astore_1
            3: monitorenter
            4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
            7: ldc          #3                  // String Method 1 start
            9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
            12: aload_1
            13: monitorexit
            14: goto         22
            17: astore_2
            18: aload_1
            19: monitorexit
            20: aload_2
            21: athrow
            22: return

https://blog.csdn.net/weixin_36759405
```

可以看到在同步块的入口和出口分别有monitorenter和monitorexit指令。当执行monitorenter指令时，线程试图获取锁也就是获取monitor（monitor对象存在于每个Java对象的对象头中，synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因）的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行monitorexit指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

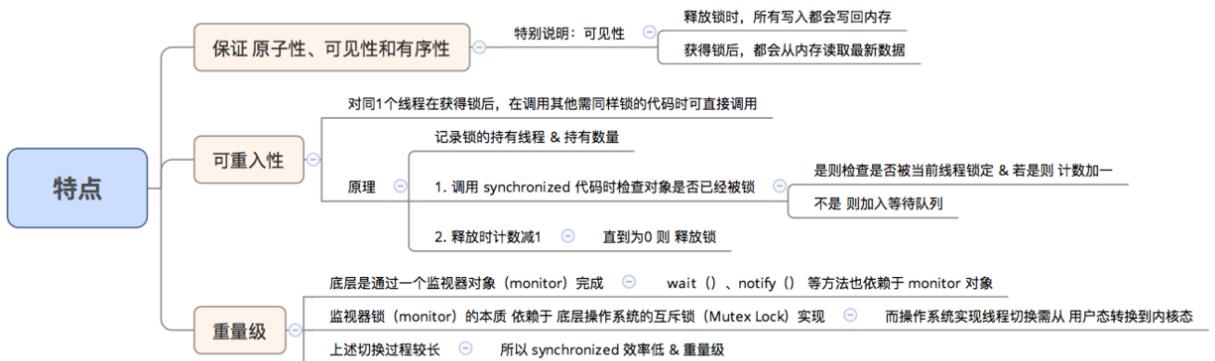
在synchronized修饰方法时是添加ACC_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

```
public synchronized void method();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
    stack=2, locals=1, args_size=1
        0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc          #3                  // String Hello World!
        5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return

LineNumberTable:
    line 5: 0
    line 6: 8

LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0       9      0  this   Lcom/paddx/test/concurrent/SynchronizedMethod;
https://blog.csdn.net/weixin_36759405
```

synchronized的特点：



https://blog.csdn.net/weixin_36759405

12.3 Synchronized 优化

从synchronized的特点中可以看到它是一种重量级锁，会涉及到操作系统状态的切换影响效率，所以JDK1.6中对synchronized进行了各种优化，为了能减少获取和释放锁带来的消耗引入了偏向锁和轻量锁。

12.3.1 偏向锁

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadId的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。

12.3.1.1 偏向锁的获取过程

- (1) 访问Mark Word中偏向锁的标识是否设置成“1”，锁标志位是否为“01”——确认为可偏向状态。
- (2) 如果为可偏向状态，判断线程ID是否指向当前线程，如果是进入步骤 (5)，否则进入步骤 (3)。
- (3) 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行 (5)；如果竞争失败，执行 (4)。
- (4) 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点(safepoint)时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。
- (5) 执行同步代码。

12.3.1.2 偏向锁的释放

偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

12.3.2 轻量锁

轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。

12.3.2.1 轻量级锁的加锁过程

(1) 在代码进入同步块时，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝。

(2) 拷贝对象头中的Mark Word复制到锁记录中。

(3) 拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指针指向object mark word。如果更新成功，则执行步骤 (3)，否则执行步骤 (4)。

(4) 如果这个更新动作成功，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态。

(5) 如果这个更新操作失败，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。

12.3.2.2 轻量级锁的解锁过程

- (1) 通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word。
- (2) 如果替换成功，整个同步过程完成。
- (3) 如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

12.3.3 其他优化

适应性自旋：在使用CAS时，如果操作失败，CAS会自旋再次尝试。由于自旋是需要消耗CPU资源的，所以如果长期自旋就白白浪费了CPU。JDK1.6加入了适应性自旋，即如果某个锁自旋很少成功获得，那么下一次就会减少自旋。

通过`--xx:+UseSpinning`参数来开启自旋（JDK1.6之前默认关闭自旋）。通过`--xx:PreBlockSpin`修改自旋次数，默认值是10次。

锁消除：锁消除指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

锁粗化：我们在写代码时推荐将同步块的作用范围限制得尽量小——只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

注意：在大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

12.4 扩展

其他控制并发/线程同步方式还有 Lock/ReentrantLock。

12.4.1 Synchronized 和 ReentrantLock 的对比

- ① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

② synchronized依赖于JVM而ReenTrantLock依赖于API

synchronized是依赖于JVM实现的，前面我们也讲到了虚拟机团队在JDK1.6为synchronized关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock是JDK层面实现的（也就是API层面，需要lock()和unlock()方法配合try/finally语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReenTrantLock比synchronized增加了一些高级功能

相比synchronized，ReenTrantLock增加了一些高级功能。主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

- ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
ReenTrantLock默认情况是非公平的，可以通过ReenTrantLoc类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
- synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition()方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。在使用notify/notifyAll()方法进行通知时，被通知的线程是由JVM选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个

Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择ReenTrantLock是一个不错的选择。

④ 性能已不是选择标准

在JDK1.6之前，synchronized的性能是比ReenTrantLock差很多。具体表示为：synchronized关键字吞吐量随线程数的增加，下降得非常严重。而ReenTrantLock基本保持一个比较稳定的水平。在JDK1.6之后JVM团队对synchronized关键字做了很多优化，性能基本能与ReenTrantLock持平。所以JDK1.6之后，性能已经不是选择synchronized和ReenTrantLock的影响因素，而且虚拟机在未来的性能改进中会更偏向于原生的synchronized，所以还是提倡在synchronized能满足你的需求的情况下，优先考虑使用synchronized关键字来进行同步！优化后的synchronized和ReenTrantLock一样，在很多地方都是用到了CAS操作。

CAS的原理是通过不断的比较内存中的值与旧值是否相同，如果相同则将内存中的值修改为新值，相比于synchronized省去了挂起线程、恢复线程的开销。

```
// CAS的操作参数  
// 内存位置（A）  
// 预期原值（B）  
// 预期新值（C）  
  
// 使用CAS解决并发的原理：  
// 1. 首先比较A、B，若相等，则更新A中的值为C、返回True；若不相等，则返回false；  
// 2. 通过死循环，以不断尝试尝试更新的方式实现并发  
  
// 伪代码如下  
public boolean compareAndSwap(long memoryA, int oldB, int newC){  
    if(memoryA.get() == oldB){  
        memoryA.set(newC);  
    }  
}
```

```
        return true;
    }
    return false;
}
```

具体使用当中CAS有个先检查后执行的操作，而这种操作在 Java 中是典型的不安全的操作，所以CAS在实际中是由C++通过调用CPU指令实现的。
具体过程：

1. CAS在Java中的体现为Unsafe类。
2. Unsafe类会通过C++直接获取到属性的内存地址。
3. 接下来CAS由C++的Atomic::cmpxchg系列方法实现。

AtomicInteger的 i++ 与 i-- 是典型的CAS应用，通过compareAndSet & 一个死循环实现。

```
private volatile int value;
/**
 * Gets the current value.
 *
 * @return the current value
 */
public final int get() {
    return value;
}
/**
 * Atomically increments by one the current value.
 *
 * @return the previous value
 */
public final int getAndIncrement() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return current;
    }
}
```

```
}

/**
 * Atomically decrements by one the current value.
 *
 * @return the previous value
 */
public final int getAndDecrement() {
    for (;;) {
        int current = get();
        int next = current - 1;
        if (compareAndSet(current, next))
            return current;
    }
}
```

以上内容引用自[学习指南](#)。

12.4.2 Synchronized 与 ThreadLocal 的对比

Synchronized 与 ThreadLocal（有关ThreadLocal的知识会在之后的博客中介绍）的比较：

1. Synchronized关键字主要解决多线程共享数据同步问题；
ThreadLocal主要解决多线程中数据因并发产生不一致问题。
2. Synchronized是利用锁的机制，使变量或代码块只能被一个线程访问。而ThreadLocal为每一个线程都提供变量的副本，使得每个线程访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。

参考链接：

[synchronized 关键字原理](#)

[Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）](#)

[Java：这是一份全面 & 详细的 Synchronized关键字 学习指南](#)

第十三节 ReentrantLock原理

ReentrantLock主要利用CAS+AQS队列来实现。它支持公平锁和非公平锁，两者的实现类似。

CAS: Compare and Swap, 比较并交换。CAS有3个操作数：内存值V、预期值A、要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。该操作是一个原子操作，被广泛的应用在Java的底层实现中。在Java中，CAS主要是由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现

ReentrantLock主要利用CAS+AQS队列来实现。它支持公平锁和非公平锁，两者的实现类似。

CAS: Compare and Swap, 比较并交换。CAS有3个操作数：内存值V、预期值A、要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。该操作是一个原子操作，被广泛的应用在Java的底层实现中。在Java中，CAS主要是由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现

AbstractQueuedSynchronizer简称AQS

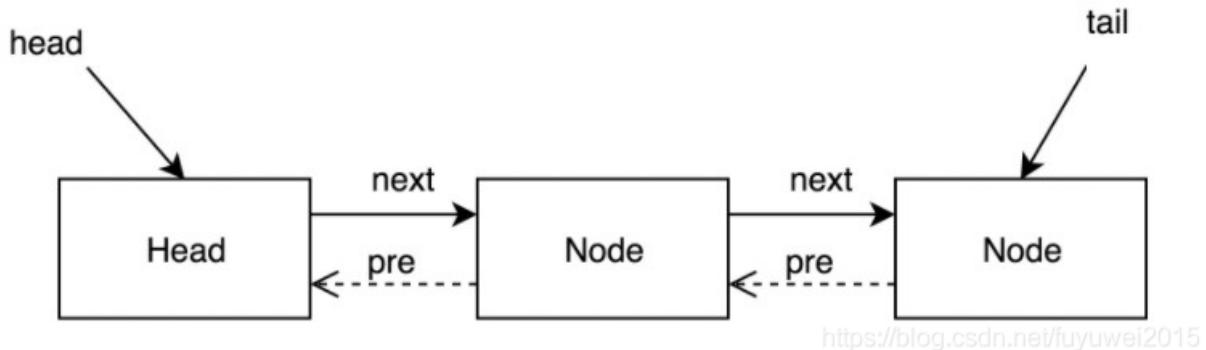
【ReentrantLock使用示例】

```
private Lock lock = new ReentrantLock();

public void test(){
    lock.lock();
    try{
        doSomeThing();
    }catch (Exception e){
        // ignored
    }finally {
        lock.unlock();
    }
}
```

【AQS】

是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock, Semaphore, CountDownLatch, ReentrantReadWriteLock, FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。



AQS使用一个FIFO的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态waitStatus

ReentrantLock的基本实现可以概括为：先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起。当锁被释放之后，排在CLH队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取；

公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁。

【lock()与unlock()实现原理】

可重入锁。可重入锁是指同一个线程可以多次获取同一把锁。

ReentrantLock和synchronized都是可重入锁。

可中断锁。可中断锁是指线程尝试获取锁的过程中，是否可以响应中断。synchronized是不可中断锁，而ReentrantLock则提供了中断功能。

公平锁与非公平锁。公平锁是指多个线程同时尝试获取同一把锁时，获取锁的顺序按照线程达到的顺序，而非公平锁则允许线程“插队”。
synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁。

CAS操作(CompareAndSwap)。CAS操作简单的说就是比较并交换。CAS操作包含三个操作数——内存位置 (V)、预期原值 (A) 和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”Java并发包(java.util.concurrent)中大量使用了CAS操作,涉及到并发的地方都调用了sun.misc.Unsafe类方法进行CAS操作。

ReentrantLock提供了两个构造器，分别是

```
public ReentrantLock() {  
    sync = new NonfairSync();  
}  
  
public ReentrantLock(boolean fair) {  
    sync = fair ? new FairSync() : new NonfairSync();  
}
```

默认构造器初始化为NonfairSync对象，即非公平锁，而带参数的构造器可以指定使用公平锁和非公平锁。由lock()和unlock的源码可以看到，它们只是分别调用了sync对象的lock()和release(1)方法。

NonfairSync

```
final void lock() {  
    if (compareAndSetState(0, 1))  
        setExclusiveOwnerThread(Thread.currentThread());  
    else  
        acquire(1);  
}
```

首先用一个CAS操作，判断state是否是0（表示当前锁未被占用），如果是0则把它置为1，并且设置当前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS操作只能保证一个线程操作成功，剩下的只能乖乖的去排队啦。

“非公平”即体现在这里，如果占用锁的线程刚释放锁，state置为0，而排队等待锁的线程还未唤醒时，新来的线程就直接抢占了该锁，那么就“插队”了。

若当前有三个线程去竞争锁，假设线程A的CAS操作成功了，拿到了锁开开心心的返回了，那么线程B和C则设置state失败，走到了else里面。我们往下看acquire。

```
public final void acquire(int arg) {  
    if (!tryAcquire(arg) &&  
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))  
        selfInterrupt();  
}
```

1. 第一步。尝试去获取锁。如果尝试获取锁成功，方法直接返回。

```
tryAcquire(arg)  
final boolean nonfairTryAcquire(int acquires) {  
    //获取当前线程  
    final Thread current = Thread.currentThread();  
    //获取state变量值  
    int c = getState();  
    if (c == 0) { //没有线程占用锁  
        if (compareAndSetState(0, acquires)) {  
            //占用锁成功，设置独占线程为当前线程  
            setExclusiveOwnerThread(current);  
            return true;  
        }  
    } else if (current == getExclusiveOwnerThread()) { //  
        //当前线程已经占用该锁  
        int nextc = c + acquires;  
        if (nextc < 0) // overflow  
            throw new Error("Maximum lock count  
exceeded");
```

```

        // 更新state值为新的重入次数
        setState(nextc);
        return true;
    }
    //获取锁失败
    return false;
}

```

非公平锁tryAcquire的流程是：检查state字段，若为0，表示锁未被占用，那么尝试占用，若不为0，检查当前锁是否被自己占用，若被自己占用，则更新state字段，表示重入锁的次数。如果以上两点都没有成功，则获取锁失败，返回false。

2. 第二步，入队。由于上文中提到线程A已经占用了锁，所以B和C执行tryAcquire失败，并且入等待队列。如果线程A拿着锁死死不放，那么B和C就会被挂起。

先看下入队的过程。先看addWaiter(Node.EXCLUSIVE)

```

/**
 * 将新节点和当前线程关联并且入队列
 * @param mode 独占/共享
 * @return 新节点
 */
private Node addwaiter(Node mode) {
    //初始化节点，设置关联线程和模式(独占 or 共享)
    Node node = new Node(Thread.currentThread(), mode);
    // 获取尾节点引用
    Node pred = tail;
    // 尾节点不为空，说明队列已经初始化过
    if (pred != null) {
        node.prev = pred;
        // 设置新节点为尾节点
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
}

```

```

    // 尾节点为空,说明队列还未初始化,需要初始化head节点并入队新节点
    enq(node);
    return node;
}

```

B、C线程同时尝试入队列，由于队列尚未初始化，tail==null，故至少会有一个线程会走到enq(node)。我们假设同时走到了enq(node)里。

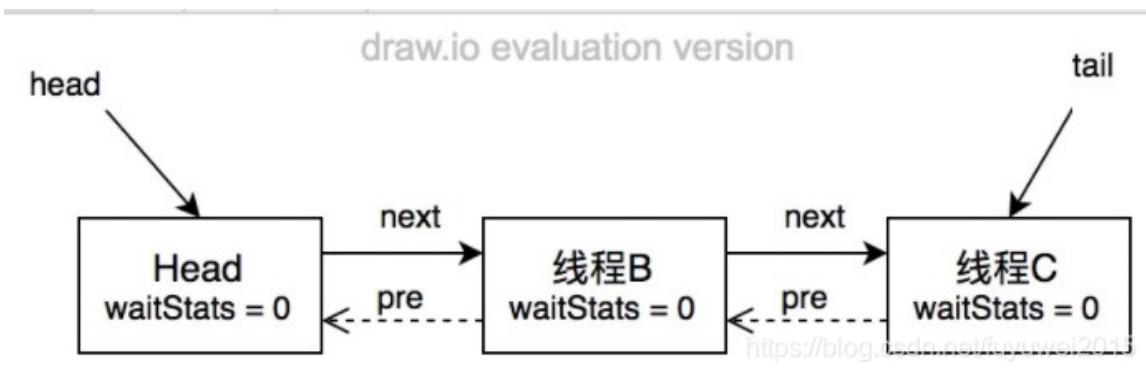
```

/**
 * 初始化队列并且入队新节点
 */
private Node enq(final Node node) {
    //开始自旋
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            // 如果tail为空,则新建一个head节点,并且tail指向head
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            // tail不为空,将新节点入队
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}

```

这里体现了经典的自旋+CAS组合来实现非阻塞的原子操作。由于compareAndSetHead的实现使用了unsafe类提供的CAS操作，所以只有一个线程会创建head节点成功。假设线程B成功，之后B、C开始第二轮循环，此时tail已经不为空，两个线程都走到else里面。假设B线程compareAndSetTail成功，那么B就可以返回了，C由于入队失败还需要第三轮循环。最终所有线程都可以成功入队。

当B、C入等待队列后，此时AQS队列如下：



3. 第三步，挂起。B和C相继执行acquireQueued(final Node node, int arg)。这个方法让已经入队的线程尝试获取锁，若失败则会被挂起。

```
/**  
 * 已经入队的线程尝试获取锁  
 */  
final boolean acquireQueued(final Node node, int arg) {  
    boolean failed = true; // 标记是否成功获取锁  
    try {  
        boolean interrupted = false; // 标记线程是否被中断过  
        for (;;) {  
            final Node p = node.predecessor(); // 获取前驱节点  
            // 如果前驱是head, 即该结点已成老二, 那么便有资格去尝试  
            // 获得锁  
            if (p == head && tryAcquire(arg)) {  
                setHead(node); // 获取成功, 将当前节点设置为  
                head节点  
                p.next = null; // 原head节点出队, 在某个时间点  
                被GC回收  
                failed = false; // 获取成功  
                return interrupted; // 返回是否被中断过  
            }  
            // 判断获取失败后是否可以挂起, 若可以则挂起  
            if (shouldParkAfterFailedAcquire(p, node) &&  
                parkAndCheckInterrupt())  
                // 线程若被中断, 设置interrupted为true  
                interrupted = true;  
        }  
    }
```

```
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

code里的注释已经很清晰的说明了acquireQueued的执行流程。假设B和C在竞争锁的过程中A一直持有锁，那么它们的tryAcquire操作都会失败，因此会走到第2个if语句中。我们再看下shouldParkAfterFailedAcquire和parkAndCheckInterrupt都做了哪些事吧。

```
/***
 * 判断当前线程获取锁失败之后是否需要挂起.
 */
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    //前驱节点的状态
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        // 前驱节点状态为signal,返回true
        return true;
    // 前驱节点状态为CANCELLED
    if (ws > 0) {
        // 从队尾向前寻找第一个状态不为CANCELLED的节点
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 将前驱节点的状态设置为SIGNAL
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

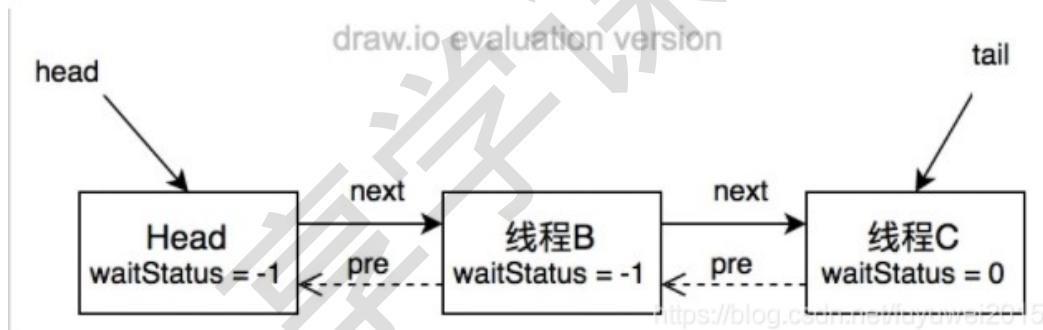
/***
 * 挂起当前线程,返回线程中断状态并重置
*/
```

```
private final boolean parkAndCheckInterrupt() {  
    LockSupport.park(this);  
    return Thread.interrupted();  
}
```

线程入队后能够挂起的前提是，它的前驱节点的状态为SIGNAL，它的含义是“Hi，前面的兄弟，如果你获取锁并且出队后，记得把我唤醒！”。所以shouldParkAfterFailedAcquire会先判断当前节点的前驱是否状态符合要求，若符合则返回true，然后调用parkAndCheckInterrupt，将自己挂起。如果不符，再看前驱节点是否>0(CANCELLED)，若是那么向前进遍历直到找到第一个符合要求的前驱，若不是则将前驱节点的状态设置为SIGNAL。

整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心挂起，需要去找个安心的挂起点，同时可以再尝试下看有没有机会去尝试竞争锁。

最终队列可能会如下图所示



unlock()

```
public void unlock() {  
    sync.release(1);  
}  
  
public final boolean release(int arg) {  
    if (tryRelease(arg)) {  
        Node h = head;  
        if (h != null && h.waitStatus != 0)  
            unparkSuccessor(h);  
        return true;  
    }  
    return false;  
}
```

如果理解了加锁的过程，那么解锁看起来就容易多了。流程大致为先尝试释放锁，若释放成功，那么查看头结点的状态是否为SIGNAL，如果是则唤醒头结点的下个节点关联的线程，如果释放失败那么返回false表示解锁失败。这里我们也发现了，每次都只唤起头结点的下一个节点关联的线程。

最后我们再看下tryRelease的执行过程

```
/**  
 * 释放当前线程占用的锁  
 * @param releases  
 * @return 是否释放成功  
 */  
protected final boolean tryRelease(int releases) {  
    // 计算释放后state值  
    int c = getState() - releases;  
    // 如果不是当前线程占用锁，那么抛出异常  
    if (Thread.currentThread() !=  
getExclusiveOwnerThread())  
        throw new IllegalMonitorStateException();  
    boolean free = false;  
    if (c == 0) {  
        // 锁被重入次数为0，表示释放成功  
        free = true;
```

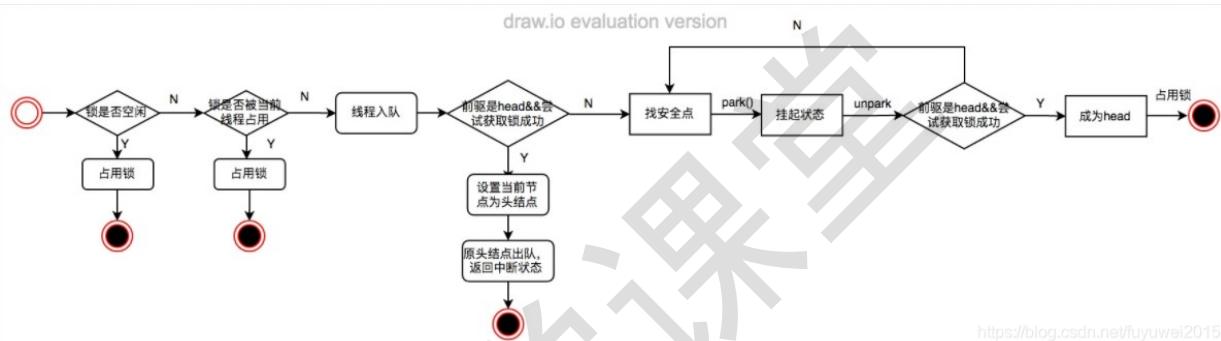
```

        // 清空独占线程
        setExclusiveOwnerThread(null);
    }
    // 更新state值
    setState(c);
    return free;
}

```

这里入参为1。tryRelease的过程为：当前释放锁的线程若不持有锁，则抛出异常。若持有锁，计算释放后的state值是否为0，若为0表示锁已经被成功释放，并且则清空独占线程，最后更新state值，返回free。

用一张流程图总结一下非公平锁的获取锁的过程。



FairSync

公平锁和非公平锁不同之处在于，公平锁在获取锁的时候，不会先去检查state状态，而是直接执行acquire(1)

超时机制

在ReentrantLock的tryLock(long timeout, TimeUnit unit) 提供了超时获取锁的功能。它的语义是在指定的时间内如果获取到锁就返回true，获取不到则返回false。这种机制避免了线程无限期的等待锁释放。那么超时的功能是怎么实现的呢？我们还是用非公平锁为例来一探究竟。

```

public boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException {
    return sync.tryAcquireNanos(1,
        unit.toNanos(timeout));
}

```

还是调用了内部类里面的方法。我们继续向前探究

```
public final boolean tryAcquireNanos(int arg, long
nanosTimeout)
    throws InterruptedException {
if (Thread.interrupted())
    throw new InterruptedException();
return tryAcquire(arg) ||
    doAcquireNanos(arg, nanosTimeout);
}
```

这里的语义是：如果线程被中断了，那么直接抛出 InterruptedException。如果未中断，先尝试获取锁，获取成功就直接返回，获取失败则进入doAcquireNanos。tryAcquire我们已经看过，这里重点看一下doAcquireNanos做了什么。

```
/***
 * 在有限的时间内去竞争锁
 * @return 是否获取成功
 */
private boolean doAcquireNanos(int arg, long
nanosTimeout)
    throws InterruptedException {
// 起始时间
long lastTime = System.nanoTime();
// 线程入队
final Node node = addWaiter(Node.EXCLUSIVE);
boolean failed = true;
try {
    // 又是自旋！
    for (;;) {
        // 获取前驱节点
        final Node p = node.predecessor();
        // 如果前驱是头节点并且占用锁成功，则将当前节点变成头结
点
        if (p == head && tryAcquire(arg)) {
            setHead(node);
            p.next = null; // help GC
            failed = false;
            return true;
        }
    }
}
} catch (InterruptedException e) {
    if (!failed)
        cancelAcquisition(node);
    throw e;
}
return false;
}
```

```
        }

        // 如果已经超时,返回false
        if (nanosTimeout <= 0)
            return false;
        // 超时时间未到,且需要挂起
        if (shouldParkAfterFailedAcquire(p, node) &&
            nanosTimeout >
            spinForTimeoutThreshold)
            // 阻塞当前线程直到超时时间到期
            LockSupport.parkNanos(this,
nanosTimeout);

        long now = System.nanoTime();
        // 更新nanosTimeout
        nanosTimeout -= now - lastTime;
        lastTime = now;
        if (Thread.interrupted())
            //相应中断
            throw new InterruptedException();
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
}
```

doAcquireNanos的流程简述为：线程先入等待队列，然后开始自旋，尝试获取锁，获取成功就返回，失败则在队列里找一个安全点把自己挂起直到超时时间过期。这里为什么还需要循环呢？因为当前线程节点的前驱状态可能不是SIGNAL，那么在当前这一轮循环中线程不会被挂起，然后更新超时时间，开始新一轮的尝试

第十四节 HashMap中的Hash冲突解决和扩容机制

14.1 关于HashMap

HashMap根据key的hash值来存储数据，HashMap最多只允许一个key为null的记录。HashMap是线程不安全的。HashMap的数据结构是：数组+链表+红黑树（JDK1.8增加了红黑树部分）。

HashMap中的几个关键属性如下：

```
//hash表
transient Node<K,V>[] table;

transient Set<Map.Entry<K,V>> entrySet;

//map中包含的key-value个数
transient int size;

//记录当前HashMap内部结构发生变化的次数，主要用于fail-fast的
//判断
transient int modCount;

//扩容阈值，capacity * load factor
//默认容量：DEFAULT_INITIAL_CAPACITY = 1 << 4
int threshold;

//负载因子
final float loadFactor;

static class Node<K,V> implements Map.Entry<K,V> {
    final int hash; //用来定位数组索引位置
    final K key;
    V value;
    Node<K,V> next; //链表的下一个node
}
```

HashMap的默认初始容量时 $1 \ll 4(16)$,HashMap的容量一定时2的n次幂。
(这是为了方便hash计算,具体可以看
<https://russxia.github.io/2019/04/29/TreeSet-LinkedHashSet-HashSet的区别.html>中关于hash函数的描述)。

HashMap使用哈希表来存储数据。Hash算法能将任意数据散列映射到有限的空间上，因此在快速查找和加密中使用广泛。为了解决哈希冲突，可以采用“开放地址法”或者“链地址法”等来解决问题。HashMap中采用的是链地址法。

14.2 关于Hash算法和Hash冲突

Hash算法:就是根据设定的Hash函数 $H(key)$ 和处理冲突方法，将一组关键字映射到一个有限的地址区间上的算法。所以Hash算法也被称为散列算法、杂凑算法。

Hash表:通过Hash算法后得到的有限地址区间上的集合。数据存放的位置和key之前存在一定的关系($H(key)=stored_value_hash$ (数据存放位置)),可以实现快速查询。与之相对的，如果数据存放位置和key之间不存在任何关联关系的集合，称之为**非Hash表**。

Hash冲突:由于用于计算的数据是无限的 $H(key)$, key 属于 $(-\infty, +\infty)$,而映射到区间是有限的，所以肯定会存在两个key: key_1, key_2 ,
 $H(key_1)=H(key_2)$, 这就是hash冲突。一般的解决Hash冲突方法有:开放定址法、再哈希法、链地址法(拉链法)、建立公共溢出区。

开放地址法

开放定址法也称为**再散列法**，基本思想就是，如果 $p=H(key)$ 出现冲突时，则以 p 为基础，再次hash， $p_1=H(p)$,如果 p_1 再次出现冲突，则以 p_1 为基础，以此类推，直到找到一个不冲突的哈希地址 p_i 。因此开放定址法所需要的hash表的长度要大于等于所需要存放的元素，而且因为存在再次hash，所以只能在删除的节点上做标记，而不能真正删除节点。

缺点:容易产生堆积问题;不适合大规模的数据存储;插入时会发生多次冲突的情况;删除时要考虑与要删除元素互相冲突的另一个元素，比较复杂。

再哈希法(双重散列，多重散列)

提供多个不同的hash函数，当`R1=H1(key1)`发生冲突时，再计算`R2=H2(key1)`，直到没有冲突为止。这样做虽然不易产生堆集，但增加了计算的时间。

链地址法(拉链法)

链地址法：将哈希值相同的元素构成一个同义词的单链表，并将单链表的头指针存放在哈希表的第*i*个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。HashMap采用的就是链地址法来解决hash冲突。(链表长度大于等于8时转为红黑树)

建立公共溢出区

将哈希表分为公共表和溢出表，当溢出发生时，将所有溢出数据统一放到溢出区。

14.3 HashMap中的处理冲突

下面是HashMap的put方法：

```
final V putVal(int hash, K key, V value, boolean  
onlyIfAbsent,  
            boolean evict) {  
    Node<K,V>[] tab; Node<K,V> p; int n, i;  
    if ((tab = table) == null || (n = tab.length) == 0)  
        //如果hash数组为空，初始化一下  
        n = (tab = resize()).length;  
    if ((p = tab[i = (n - 1) & hash]) == null) //计算落在  
    hash桶的位置，如果当前桶为空，直接新增节点  
        tab[i] = newNode(hash, key, value, null);  
    else { //当前桶存在元素  
        Node<K,V> e; K k;  
        if (p.hash == hash &&  
            ((k = p.key) == key || (key != null &&  
            key.equals(k))))  
            //如果key已经存在，替换元素  
            e = p;  
        else if (p instanceof TreeNode) //如果当前是树结构了  
        (不是链表了)，向树上添加元素
```

```
        e = ((TreeNode<K,V>)p).putTreeVal(this, tab,
hash, key, value);
    else { //当前结构依然时链表，遍历链表，直到末尾或者找到
key相同的元素替换
        for (int binCount = 0; ; ++binCount) {
            //到达末尾，新增元素，如果链表长度达到8，转为红黑
树
            if ((e = p.next) == null) {
                p.next = newNode(hash, key, value,
null);
                if (binCount >= TREEIFY_THRESHOLD -
1) // -1 for 1st
                    treeifyBin(tab, hash);
                break;
            }
            //遍历链表的过程中，发现了有key相同的元素，直接替
换，然后break
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null
&& key.equals(k))))
                break;
            p = e;
        }
    }
    if (e != null) { //如果是已经存在的元素，判断是否替换
(onlyIfAbsent)
        v oldvalue = e.value;
        if (!onlyIfAbsent || oldvalue == null)
            e.value = value;
        afterNodeAccess(e);
        return oldvalue;
    }
    ++modCount;
    //如果容量超过阈值，扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
```

```
    return null;  
}
```

#####

14.4HashMap中的扩容机制

当HashMap中元素的个数达到了阈值，(capacity * load factor)，
HashMap中，元素存放的位置与hash数组的个数是有关系的(`tab[i = (n - 1) & hash]`)，所以当发生扩容时，hash数组的个数发生了变化，
这个时候，元素也需要重新进行hash计算。

```
final Node<K,V>[] resize() {  
    Node<K,V>[] oldTab = table;  
    //老的容量和阈值  
    int oldCap = (oldTab == null) ? 0 : oldTab.length;  
    int oldThr = threshold;  
    //计算新的容量和阈值  
    int newCap, newThr = 0;  
    if (oldCap > 0) {  
        if (oldCap >= MAXIMUM_CAPACITY) {  
            threshold = Integer.MAX_VALUE;  
            return oldTab;  
        }  
        else if ((newCap = oldCap << 1) <  
MAXIMUM_CAPACITY &&  
                oldCap >= DEFAULT_INITIAL_CAPACITY)  
            newThr = oldThr << 1; // double threshold  
    }  
    else if (oldThr > 0) // initial capacity was placed  
in threshold  
        newCap = oldThr;  
    else { // zero initial threshold  
signifies using defaults  
        newCap = DEFAULT_INITIAL_CAPACITY;  
        newThr = (int)(DEFAULT_LOAD_FACTOR *  
DEFAULT_INITIAL_CAPACITY);
```

```

    }

    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft <
        (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    }

    threshold = newThr;
    //构建新的hash桶
    @SuppressWarnings({"rawtypes", "unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[][])new
    Node[newCap];
    table = newTab;
    if (oldTab != null) {
        //遍历原由的hash桶, copy元素到新的里面
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null; //原来的hash桶置空
                if (e.next == null) //原来位置上只有一个元素
                    //没有链表和树),直接放到新的位置
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode) //如果是树
                    //状结构, 单独处理
                    ((TreeNode<K,V>)e).split(this,
                    newTab, j, oldCap);
                else { // preserve order 这里的是链表元素
                    Node<K,V> loHead = null, loTail =
                    null; //原来index位置
                    Node<K,V> hiHead = null, hiTail =
                    null; //新的index位置
                    Node<K,V> next;
                    //循环处理链表元素, 判断元素是放在原index位
                    //置还是放在新index位置
                    do {
                        next = e.next;
                        //放在原index的位置的元素, loHead拿到
                        //链表头, loTail拿到链表尾

```

```
        if ((e.hash & oldCap) == 0) {
            if (loTail == null)
                loHead = e;
            else
                loTail.next = e;
            loTail = e;
        }
        //放在新index位置的元素，hiHead拿到链表头，•hiTail拿到链表尾
    } else {
        if (hiTail == null)
            hiHead = e;
        else
            hiTail.next = e;
        hiTail = e;
    }
} while ((e = next) != null);
//放入原index处
if (loTail != null) {
    loTail.next = null;
    newTab[j] = loHead;
}
//放入新index处
if (hiTail != null) {
    hiTail.next = null;
    newTab[j + oldCap] = hiHead;
}
}
}
}
return newTab;
}
```

其中比较有意思的是，由于put时计算hash数组角标是通过 `i = (n - 1) & hash` 计算的，其中n是的2的x次幂，扩容时，容量变为原来的两倍，`n-1 == (n-1)<<1 & 1`，所以hash桶中的元素，要么还在原来的index位置，要么在 `oldIndex+oldCap` 的位置。所以放入新的index时，元素下表：`newTab[j + oldCap]`。

关于treenode，HashMap在扩容时，如果当前节点是棵树，先还是和链表的一样，将所有元素分为两批(一批还在当前index，另一批在新的index位置，此处传入的bit=oldCap，所以和链表的还是蛮类似的)。分成的两组，分别判断当前元素个数是否小于阈值，如果是，还原成链表。

```
final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit) {
    TreeNode<K,V> b = this;
    // Relink into lo and hi lists, preserving order
    TreeNode<K,V> loHead = null, loTail = null;
    TreeNode<K,V> hiHead = null, hiTail = null;
    int lc = 0, hc = 0;
    for (TreeNode<K,V> e = b, next; e != null; e =
next) {
        next = (TreeNode<K,V>)e.next;
        e.next = null;
        if ((e.hash & bit) == 0) {
            if ((e.prev = loTail) == null)
                loHead = e;
            else
                loTail.next = e;
            loTail = e;
            ++lc;
        }
        else {
            if ((e.prev = hiTail) == null)
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
            ++hc;
        }
    }
    if (lc > threshold)
        map.rehash();
}
```

```
        }

    if (loHead != null) {
        if (lc <= UNTREEIFY_THRESHOLD)
            tab[index] = loHead.untreeify(map);
        else {
            tab[index] = loHead;
            if (hiHead != null) // (else is already
treeified)
                loHead.treeify(tab);
        }
    }

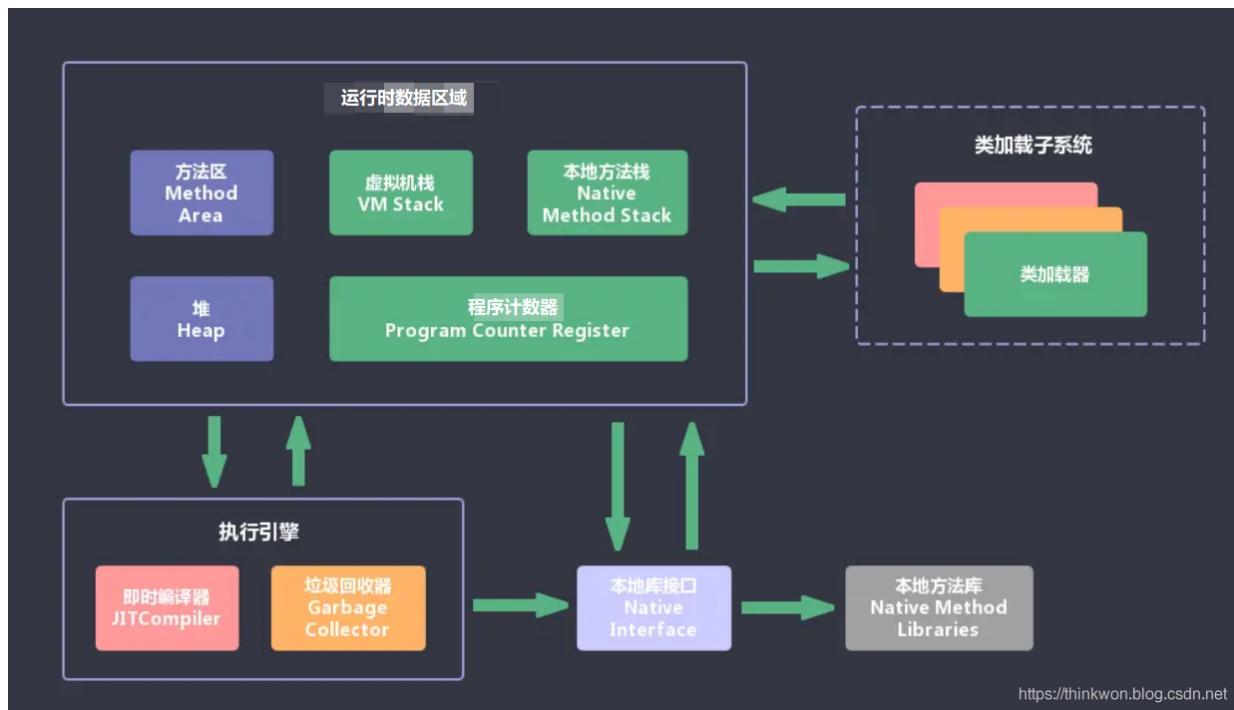
    if (hiHead != null) {
        if (hc <= UNTREEIFY_THRESHOLD)
            tab[index + bit] = hiHead.untreeify(map);
        else {
            tab[index + bit] = hiHead;
            if (loHead != null)
                hiHead.treeify(tab);
        }
    }
}
```

第十五节 JVM 常见面试题

15.1 Java 虚拟机(JVM)面试题 (2022最新版)

15.1.1 Java 内存区域

说一下 JVM 的主要组成部分及其作用？



JVM包含两个子系统和两个组件，两个子系统为Class loader(类装载)、Execution engine(执行引擎)；两个组件为Runtime data area(运行时数据区)、Native Interface(本地接口)。

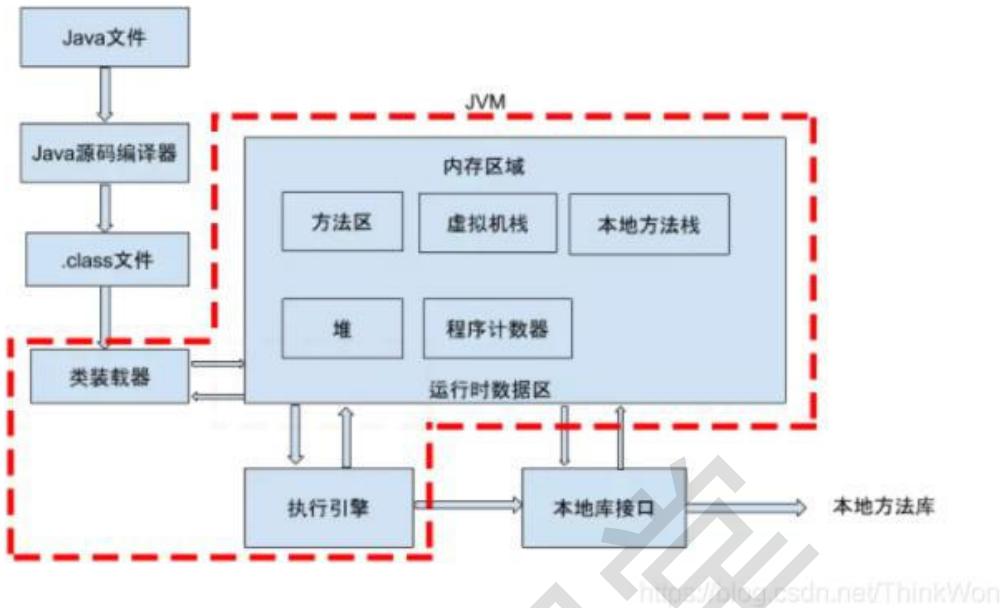
- Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。
- Execution engine (执行引擎)：执行classes中的指令。
- Native Interface(本地接口)：与native libraries交互，是其它编程语言交互的接口。
- Runtime data area(运行时数据区域)：这就是我们常说的JVM的内存。

作用：首先通过编译器把 Java 代码转换成字节码，类加载器 (ClassLoader) 再把字节码加载到内存中，将其放在运行时数据区 (Runtime data area) 的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎 (Execution Engine)，将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口 (Native Interface) 来实现整个程序的功能。

下面是Java程序运行机制详细说明

Java程序运行机制步骤

- 首先利用IDE集成开发工具编写Java源代码，源文件的后缀为.java；
- 再利用编译器(javac命令)将源代码编译成字节码文件，字节码文件的后缀名为.class；
- 运行字节码的工作是由解释器(java命令)来完成的。

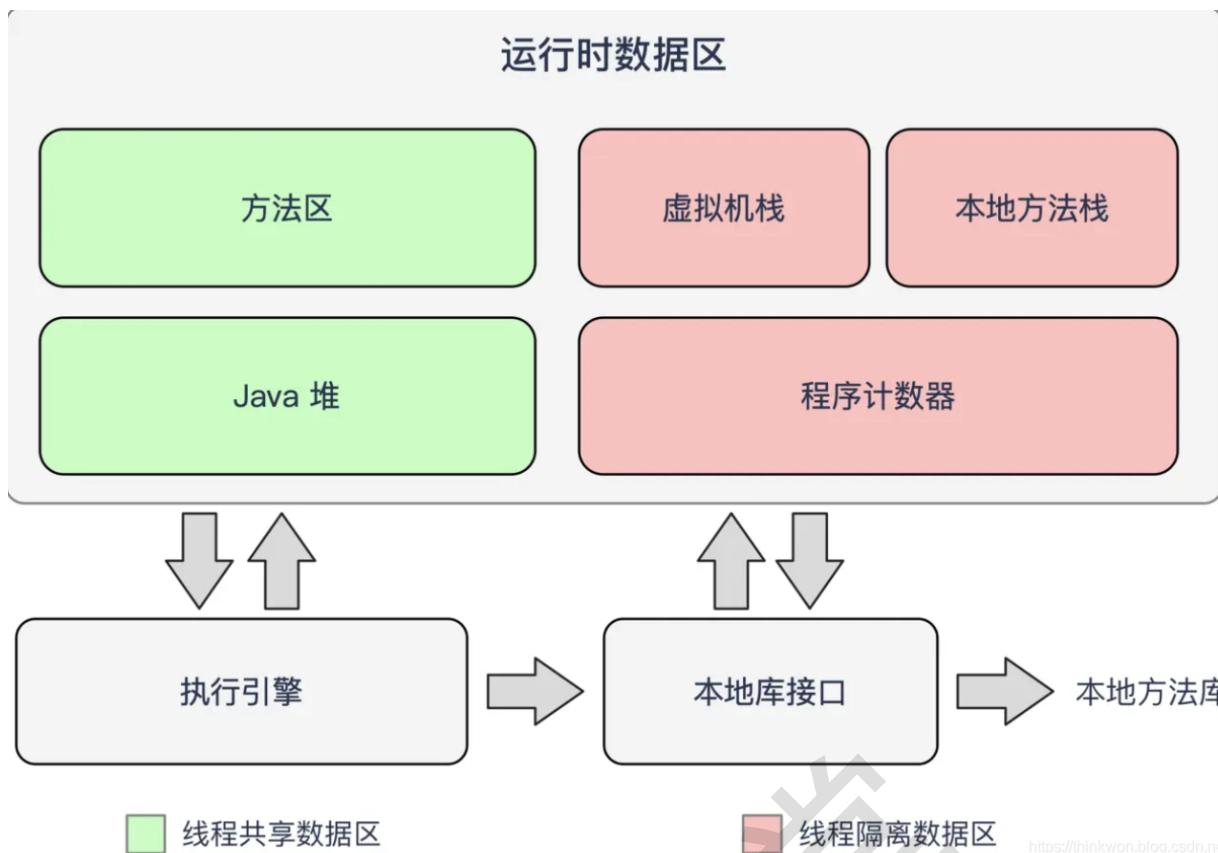


从上图可以看，java文件通过编译器变成了.class文件，接下来类加载器又将这些.class文件加载到JVM中。

其实可以一句话来解释：类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

说一下 JVM 运行时数据区

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销毁。Java 虚拟机所管理的内存被划分为如下几个区域：



不同虚拟机的运行时数据区可能略微有所不同，但都会遵从 Java 虚拟机规范，Java 虚拟机规范规定的区域分为以下 5 个部分：

- 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；
- Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；
- 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；
- Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；
- 方法区（Method Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

深拷贝和浅拷贝

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，

深拷贝 (deepCopy) 是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，

使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。

深复制：在计算机中开辟一块新的内存地址用于存放复制的对象。

说一下堆栈的区别？

物理地址

堆的物理地址分配对对象是不连续的。因此性能慢些。在GC的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）

栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

内存分别

堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS：

1. 静态变量放在方法区
2. 静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

队列和栈是什么？有什么区别？

队列和栈都是被用来预存储数据的。

- 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
- 可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
- 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO），即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

15.1.2 HotSpot虚拟机对象探秘

对象的创建

说到对象的创建，首先让我们看看 Java 中提供的几种对象创建方式：

| Header | 解释 |
|------------------------------|----------|
| 使用new关键字 | 调用了构造函数 |
| 使用Class的newInstance方法 | 调用了构造函数 |
| 使用Constructor类的newInstance方法 | 调用了构造函数 |
| 使用clone方法 | 没有调用构造函数 |
| 使用反序列化 | 没有调用构造函数 |

下面是对象创建的主要流程:



虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。类加载通过后，接下来分配内存。若Java堆中内存是绝对规整的，使用“指针碰撞”方式分配内存；如果不是规整的，就从空闲列表中分配，叫做“空闲列表”方式。划分内存时还需要考虑一个问题-并发，也有两种方式：CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码...)，最后执行<init>方法。

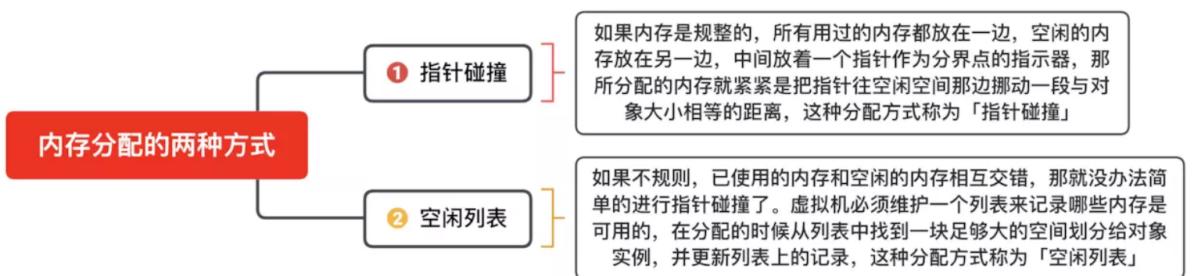
为对象分配内存

类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java堆是否规整，有两种方式：

- **指针碰撞**：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。

- 空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

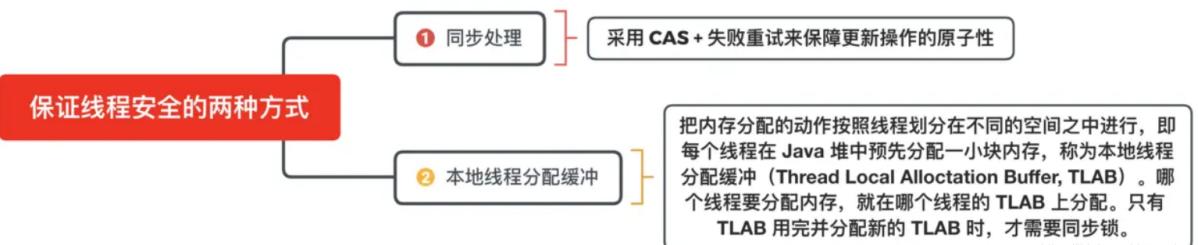
选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。



处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案：

- 对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；
- 把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）。哪个线程要分配内存，就在哪个线程的 TLAB 上分配。只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁。通过-XX:+/-UserTLAB参数来设定虚拟机是否使用TLAB。



对象的访问定位

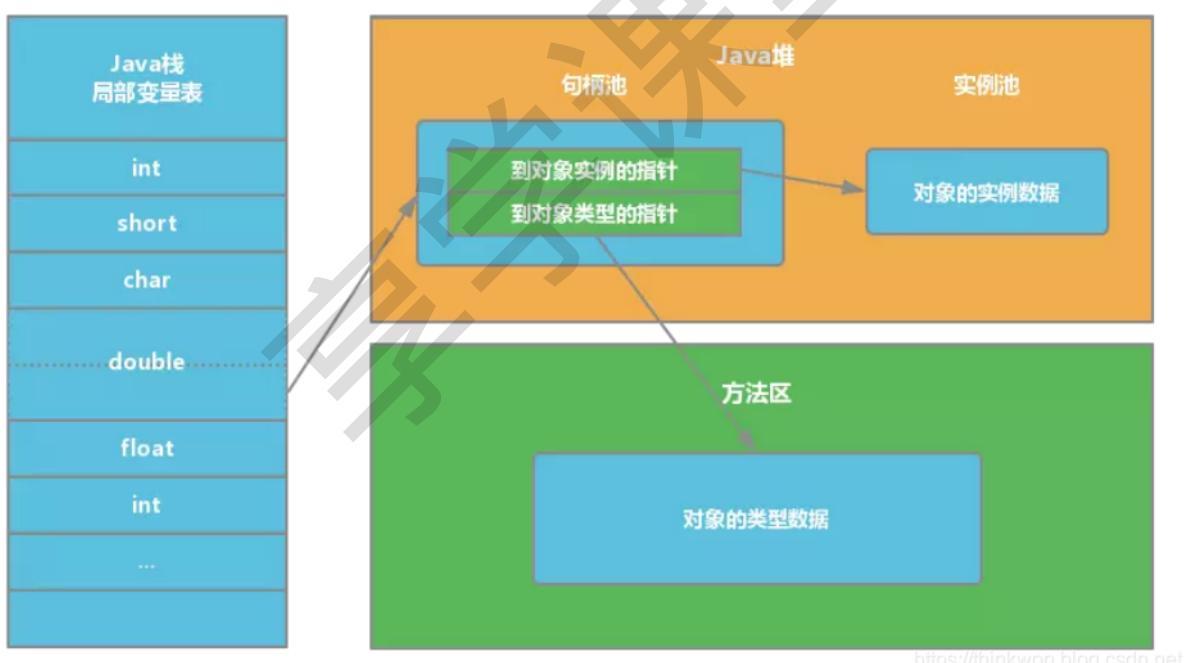
Java 程序需要通过 JVM 栈上的引用访问堆中的具体对象。对象的访问方式取决于 JVM 虚拟机的实现。目前主流的访问方式有 **句柄** 和 **直接指针** 两种方式。

指针：指向对象，代表一个对象在内存中的起始地址。

句柄：可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

句柄访问

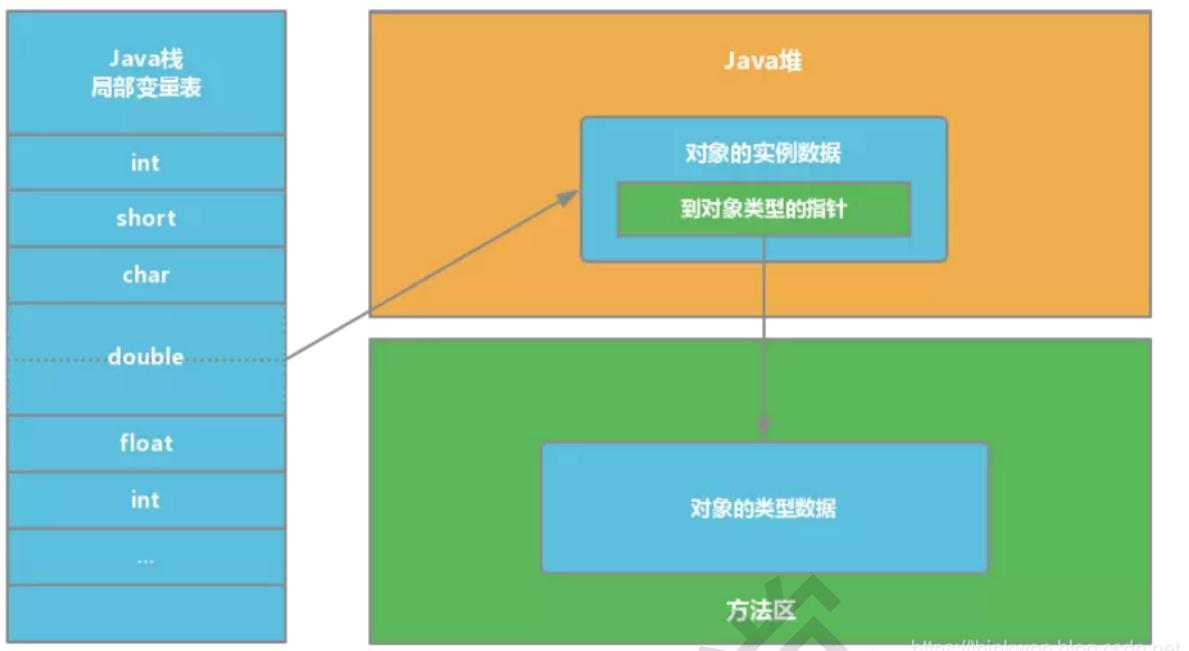
Java 堆中划分出一块内存来作为**句柄池**，引用中存储对象的**句柄地址**，而句柄中包含了**对象实例数据与对象类型数据各自的具体地址信息**，具体构造如下图所示：



优势：引用中存储的是**稳定的**句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变**句柄中的实例数据指针**，而**引用本身**不需要修改。

直接指针

如果使用直接指针访问，引用中存储的直接就是对象地址，那么 Java 堆对对象内部的布局中就必须考虑如何放置访问类型数据的相关信息。



<https://thinkworld.blog.cdn.net>

优势：速度更快，节省了一次指针定位的时间开销。由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是非常可观的执行成本。HotSpot 中采用的就是这种方式。

15.1.3 内存溢出异常

Java 会存在内存泄漏吗？请简单描述

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java 是有 GC 垃圾回收机制的，也就是说，不再被使用的对象，会被 GC 自动回收掉，自动从内存中清除。

但是，即使这样，Java 也还是存在着内存泄漏的情况，java 导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 java 中内存泄露的发生场景。

15.1.4 垃圾收集器

简述Java垃圾回收机制

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

GC是什么？为什么要GC

GC 是垃圾收集的意思（Garbage Collection），内存处理是编程人员容易出现问题的地方，忘记或者错误的内存

回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动

回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

垃圾回收的优点和原理。并考虑2种回收机制

java语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时不再考虑内存管理的问题。

由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。

垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

垃圾回收有分代复制垃圾回收、标记垃圾回收、增量垃圾回收。

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。

通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是“可达的”，哪些对象是“不可达的”。当GC确定一些对象为“不可达”时，GC就有责任回收这些内存空间。

可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

Java 中都有哪些引用类型？

- 强引用：发生 gc 的时候不会被回收。
- 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
- 弱引用：有用但不是必须的对象，在下一次GC时会被回收。
- 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

怎么判断对象是否可以被回收？

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

一般有两种方法来判断：

- 引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；
- 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

在Java中，对象什么时候可以被垃圾回收

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。

JVM中的永久代中会发生垃圾回收吗

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区

(译者注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

说一下 JVM 有哪些垃圾回收算法？

- 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。
- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。
- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

标记-清除算法

标记无用对象，然后进行清除回收。

标记-清除算法（Mark-Sweep）是一种常见的基础垃圾收集算法，它将垃圾收集分为两个阶段：

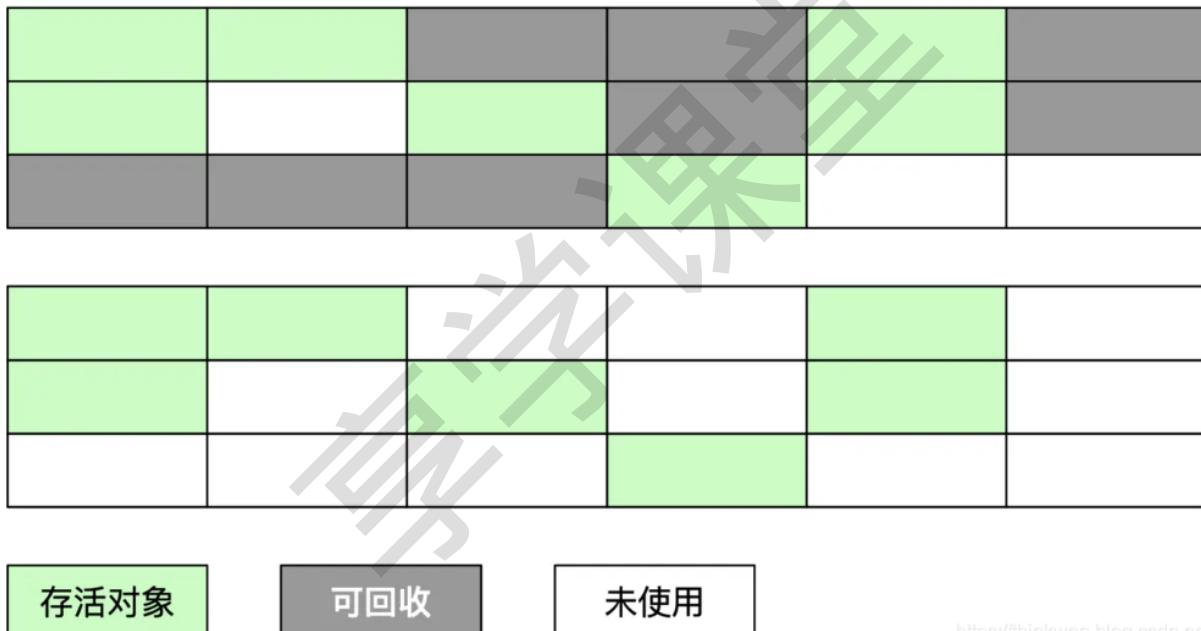
- 标记阶段：标记出可以回收的对象。
- 清除阶段：回收被标记的对象所占用的空间。

标记-清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法的基础上进行改进的。

优点：实现简单，不需要对象进行移动。

缺点：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。

标记-清除算法的执行的过程如下图所示



<https://thinkwon.blog.csdn.net>

复制算法

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

复制算法的执行过程如下图所示



标记-整理算法

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题。

缺点：仍需要进行局部对象移动，一定程度上降低了效率。

标记-整理算法的执行过程如下图所示

| | | | | | |
|--|--|--|--|--|--|
| | | | | | |
| | | | | | |
| | | | | | |

| | | | | | |
|--|--|--|--|--|--|
| | | | | | |
| | | | | | |
| | | | | | |

存活对象

可回收

未使用

<https://thinkwon.blog.csdn.net>

分代收集算法

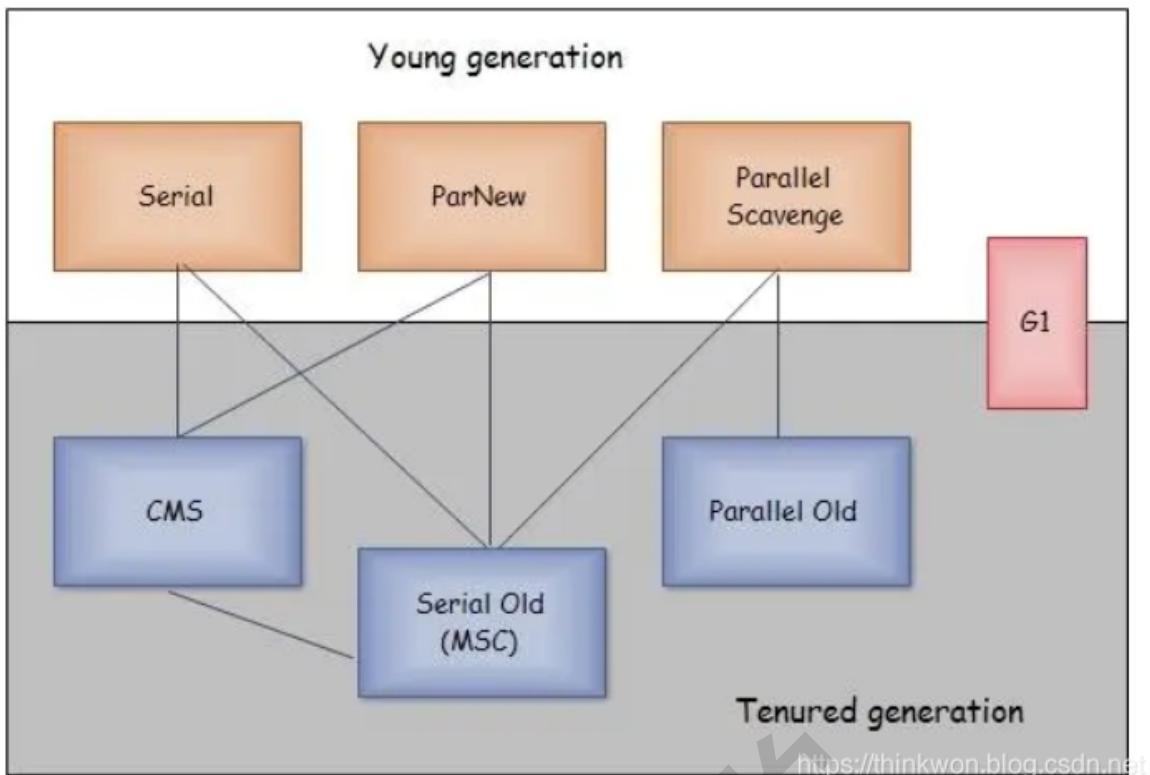
当前商业虚拟机都采用**分代收集**的垃圾收集算法。分代收集算法，顾名思义是根据对象的**存活周期**将内存划分为几块。一般包括**年轻代、老年代**和**永久代**，如图所示：



<https://thinkwon.blog.csdn.net>

说一下 JVM 有哪些垃圾回收器？

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。



<https://thinkwon.blog.csdn.net>

- **Serial**收集器 (复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
- **ParNew**收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
- **Parallel Scavenge**收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
- **Serial Old**收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；
- **Parallel Old**收集器 (标记-整理算法): 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；
- **CMS(Concurrent Mark Sweep)收集器 (标记-清除算法)** : 老年代并行收集器，以获取最短回收停顿时间为为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
- **G1(Garbage First)收集器 (标记-整理算法)**: Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

详细介绍一下 CMS 垃圾回收器？

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

新生代垃圾回收器和老年代垃圾回收器都有哪些？有什么区别？

- 新生代回收器：Serial、ParNew、Parallel Scavenge
- 老年代回收器：Serial Old、Parallel Old、CMS
- 整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

简述分代垃圾回收器是怎么工作的？

分代回收器有两个分区：老年代和新生代，新生代默认的空间占比总空间的 1/3，老年代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
- 清空 Eden 和 From Survivor 分区；
- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老年代。大对象也会直接进入老年代。

老生代当空间占用到达某个值之后就会触发全局垃圾回收，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

15.1.5 内存分配策略

简述java内存分配与回收策率以及Minor GC和Major GC

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

- **Minor GC** 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
- **Major GC/Full GC** 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15）就会被晋升到老年代。

15.1.6 虚拟机类加载机制

简述java类加载机制？

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的 java 类型。

描述一下JVM加载Class文件的原理机制

Java 中的所有类，都需要由类加载器装载到 JVM 中才能运行。类加载器本身也是一个类，而它的工作就是把 class 文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种：

1. 隐式装载，程序在运行过程中当碰到通过 new 等方式生成对象时，隐式调用类装载器加载对应的类到 jvm 中，

2. 显式装载，通过 class.forName() 等方法，显式加载需要的类

Java 类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类（像是基类）完全加载到 jvm 中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

什么是类加载器，类加载器有哪些？

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器：

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器 (system class loader) : 它根据 Java 应用的类路径 (CLASSPATH) 来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

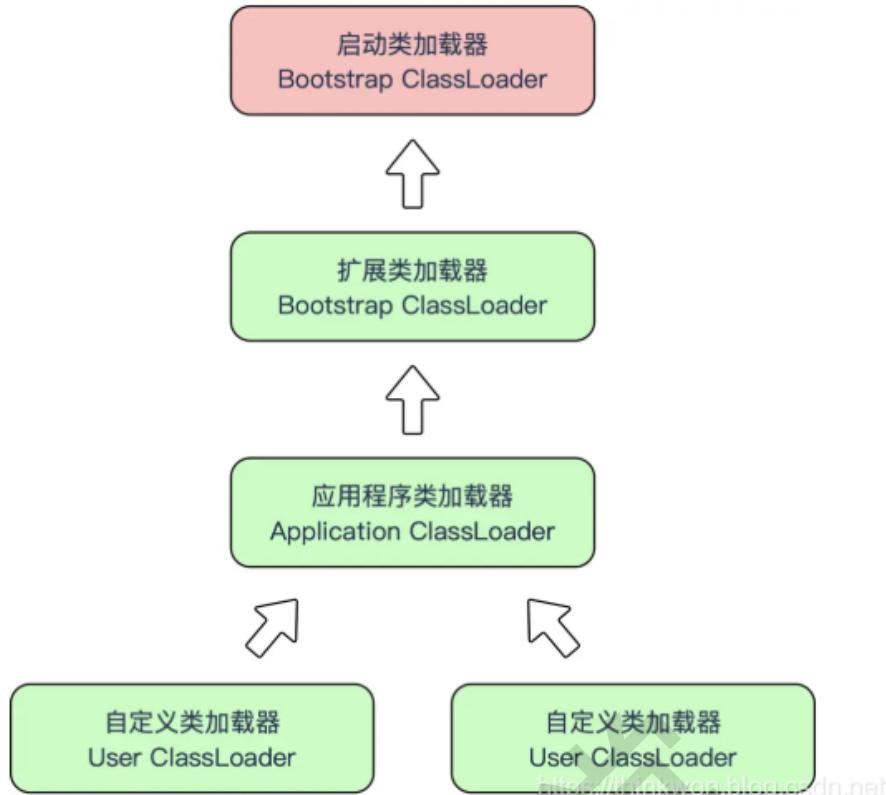
说一下类装载的执行过程？

类装载分为以下 5 个步骤：

- 加载：根据查找路径找到相应的 class 文件然后导入；
- 验证：检查加载的 class 文件的正确性；
- 准备：给类中的静态变量分配内存空间；
- 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
- 初始化：对静态变量和静态代码块执行初始化工作。

什么是双亲委派模型？

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。



类加载器分类：

- 启动类加载器 (Bootstrap ClassLoader)，是虚拟机自身的一部分，用来加载Java_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；
- 其他类加载器：
- 扩展类加载器 (Extension ClassLoader)：负责加载\lib\ext目录或 Java. ext. dirs系统变量指定的路径中的所有类库；
- 应用程序类加载器 (Application ClassLoader)。负责加载用户类路径 (classpath) 上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

15.1.7JVM调优

说一下 JVM 调优的工具?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

- jconsole：用于对 JVM 中的内存、线程和类等进行监控；
- jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

常用的 JVM 调优的参数都有哪些？

- -Xms2g：初始化堆大小为 2g；
- -Xmx2g：堆最大内存为 2g；
- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
- -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
- -XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；
- -XX:+UseParallelOldGC：指定使用 ParNew + Parallel Old 垃圾回收器组合；
- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
- -XX:+PrintGC：开启打印 gc 信息；
- -XX:+PrintGCDetails：打印 gc 详细信息。

15.2 JVM 面试题汇总

15.2.1JVM内存

- 1、JVM方法区存储内容 是否会动态扩展，是否会出现内存溢出，出现的原因有哪些？
- 2、JVM内存分为哪些区域？每个区域都存储哪些数据？一个对象从创建到销毁都是怎么在这些部分里存活和转移的？内存的哪些部分会参与GC的回收？
- 3、JVM内存分代，Java 8的内存分代改进

15.2.2 垃圾收集

- 1、常见的垃圾回收算法，Hotspot虚拟机中每种收集器使用的是哪些算法，各种算法的优缺点
- 2、列举jvm堆溢出和栈溢出的场景
- 3、JVM垃圾回收机制，何时触发MinorGC、FullGC
- 4、新生代和老生代的内存回收策略
- 5、什么时候一个对象会被GC？为什么要在这种时候对象才会被GC？GC策略都有哪些分类？这些策略分别都有什么优劣势？都适用于什么场景？举个实际的场景，让你选择一个GC策略？为什么要选择这个策略？
- 6、用什么方式可以查看是哪个类占用资源较多
- 7、JVM参数都有哪些？GC日志参数、设置各区域大小的参数
- 8、常用垃圾收集器的优缺点，G1收集器
- 9、测试环境、生产环境都是怎么配置堆栈内部各分区大小的？为什么这么配置？
- 10、jvm调优有什么思路；
- 11、jvm中的栈区一般分配多大，为什么
- 12、Eden和Survivor的比例分配默认是多少
- 13、java8中jvm做了哪两个主要的调整
- 14、在tomcat中如何配置JVM各个区域的内存大小？
- 15、你有没有遇到过OutOfMemory问题？你是怎么来处理这个问题的？处理过程中有哪些收获？
- 16、内存溢出和内存泄漏分别是什么？有什么区别？
- 17、内存溢出有哪些种类？分别是什么原因引起的？
- 18、怎样解决内存溢出问题（使用什么工具快速定位）
- 19、如何查看垃圾回收日志
- 20、当一个Java程序响应很慢时如何查找问题、
- 21、当一个Java程序频繁FullGC时如何解决问题、
- 22、当一个Java应用发生OutOfMemory时该如何解决、
- 23、如何判断是否出现死锁
- 24、如何判断是否存在内存泄露
- 25、CPU使用率高居不下/系统无响应或响应慢，该怎么解决（使用什么工具？）

15.2.3类加载

- 1、Java类加载的顺序是什么，每个步骤都做了哪些工作
- 2、Java的类加载器都有哪些？每个类加载器都加载哪些类？这些类加载之间的父子关系是怎样的？双亲委派模型是什么？有什么好处？什么情况下会破坏双亲委派模型？
- 3、如何自定义一个类加载器？你使用过哪些或者你在什么场景下需要一个自定义的类加载器吗？自己的类加载器和Java自带的类加载器关系如何处理？
- 4、ClassLoader源码看过没？loadClass方法、findClass方法、findLoadedClass方法的作用。
- 5、编译与反编译：javac、javap、jad、CRF的用法
- 6、JVM的编译优化、JIT即时编译器
- 7、oop-klass模型、对象头
- 8、Java.lang.NoSuchFieldError错误可能在什么阶段抛出

15.2.4JVM内存模型

- 1、Java的内存模型是怎么设计的？为什么要这么设计？结合内存模型的设计谈谈volatile关键字的作用？
- 2、Java内存可见性
- 3、对Java内存模型的理解，以及其在并发中的应用
- 4、指令重排序，内存栅栏等
- 5、内存可见性、重排序、顺序一致性、volatile、锁、final

15.3 JVM方法区存储内容 是否会动态扩展 是否会出现内存溢出 出现的原因有哪些

15.4 如何解决同时存在的 对象创建和对象回收问题？

15.5 JVM中最大堆大小有没有限制？

15.6 Java运行时数据区域，导致内存溢出的原因

15.7 Java中一个对象从创建到销毁的过程和JVM类加载过程

```
Student stu = new Student("zhangsan");
```

- 1、用户创建了一个Student对象，运行时JVM首先会去方法区寻找该对象的类型信息，没有则使用类加载器classloader将Student.class字节码文件加载至内存中的方法区，并将Student类的类型信息存放至方法区。
- 2、接着JVM在堆中为新的Student实例分配内存空间，这个实例持有指向方法区的Student类型信息的引用，引用指的是类型信息在方法区中的内存地址。
- 3、在此运行的JVM进程中，会首先起一个线程跑该用户程序，而创建线程的同时也创建了一个虚拟机栈，虚拟机栈用来跟踪线程运行中的一系列方法调用的过程，每调用一个方法就会创建并往栈中压入一个栈帧，栈帧用来存储方法的参数，局部变量和运算过程的临时数据。上面程序中的stu是对Student的引用，就存放于栈中，并持有指向堆中Student实例的内存地址。

4、JVM GC

JVM类加载过程

加载->验证->准备->解析->初始化->使用->卸载

类加载顺序

在类中，加载顺序为：

- 1.首先加载父类的静态字段或者静态语句块
- 2.子类的静态字段或静态语句块
- 3.父类普通变量以及语句块
- 4.父类构造方法被加载
- 5.子类变量或者语句块被加载
- 6.子类构造方法被加载

第十六节 JVM内存结构

16.1 JVM内存结构【方法区+虚拟机栈+本地方法栈+程序计数器+堆】

16.1.1 java内存组成介绍：堆(Heap)和非堆(Non-heap)内存

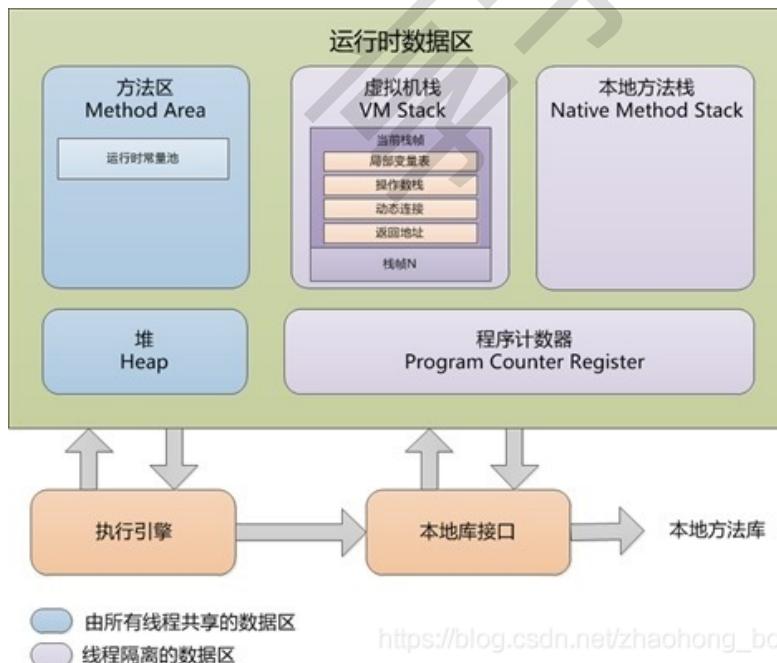
按照官方的说法：“Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。

堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。

可以看出JVM主要管理两种类型的内存：堆和非堆。简单来说堆就是 Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存(如IT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

16.1.2 JVM内存区域模型

JVM内存结构图：



1.方法区

也称“永久代”、“非堆”，它用于存储虚拟机加载的类信息、常量、静态变量、**是各个线程共享的内存区域**。默认最小值为16MB，最大值为64MB，可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

运行时常量池：是方法区的一部分，其中的主要内容来自于JVM对Class的加载。

Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种符号引用，这部分内容将在类加载后放到方法区的运行时常量池中。

2. 虚拟机栈

描述的是java 方法执行的内存模型：每个方法被执行的时候 都会创建一个“栈帧”用于存储局部变量表(包括参数)、操作栈、方法出口等信息。每个方法被调用到执行完的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。**声明周期与线程相同，是线程私有的。**

局部变量表存放了编译器可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用(引用指针，并非对象本身)，其中64位长度的long和double类型的数据会占用2个局部变量的空间，其余数据类型只占1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量是完全确定的，在运行期间栈帧不会改变局部变量表的大小空间。

3.本地方法栈

与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法服务。

4.堆

也叫做java 堆、GC堆是java虚拟机所管理的内存中最大的一块内存区域，也是被**各个线程共享的内存区域**，在JVM启动时创建。

该内存区域存放了对象实例及数组(所有new的对象)。其大小通过-Xms(最小值)和-Xmx(最大值)参数设置，-Xms为JVM启动时申请的最小内存，默认认为操作系统物理内存的1/64但小于1G，-Xmx为JVM可申请的最大内存，默认认为物理内存的1/4但小于1G，默认当空余堆内存小于40%时，

JVM会增大Heap到-Xmx指定的大小，可通过-XX:MinHeapFreeRatio=来指定这个比例；当空余堆内存大于70%时，JVM会减小heap的大小到-Xms指定的大小，可通过XX:MaxHeapFreeRatio=来指定这个比例，对于运行系统，为避免在运行时频繁调整Heap的大小，通常-Xms与-Xmx的值设成一样。

由于现在收集器都是采用分代收集算法，堆被划分为新生代和老年带。新生代主要存储新创建的对象和尚未进入老年带的对象。老年带存储经过多次新生代GC(Minor GC)任然存活的对象。

新生代：程序新创建的对象都是从新生代分配内存，新生代由Eden Space和两块相同大小的Survivor

Space(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRatio来调整Eden Space及Survivor Space的大小。

老年带：

用于存放经过多次新生代GC任然存活的对象，例如缓存对象，新建的对象也有可能直接进入老年带，主要有两种情况：

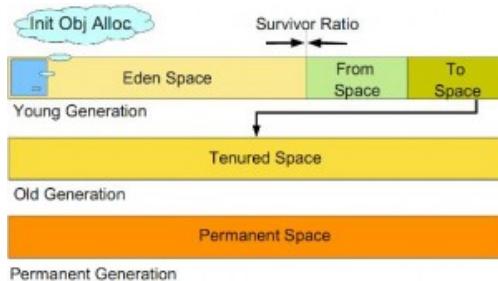
①.大对象，可通过启动参数设置-

XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年带分配。

②.大的数组对象，且数组中无引用外部对象。

老年带所占的内存大小为-Xmx对应的值减去-Xmn对应的值。

堆内存示意图



| | |
|------------------|----------------------------|
| Young Generation | 即图中的Eden + From Space + To |
| Space | |
| Eden | 存放新生的对象 |
| Survivor Space | 有两个，存放每次垃圾回收后存活的对象 |
| Old Generation | Tenured Generation 即图中的old |
| Space | |
| | 主要存放应用程序中生命周期长的存活对象 |

5.程序计数器

是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。

16.1.3直接内存

直接内存并不是虚拟机内存的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

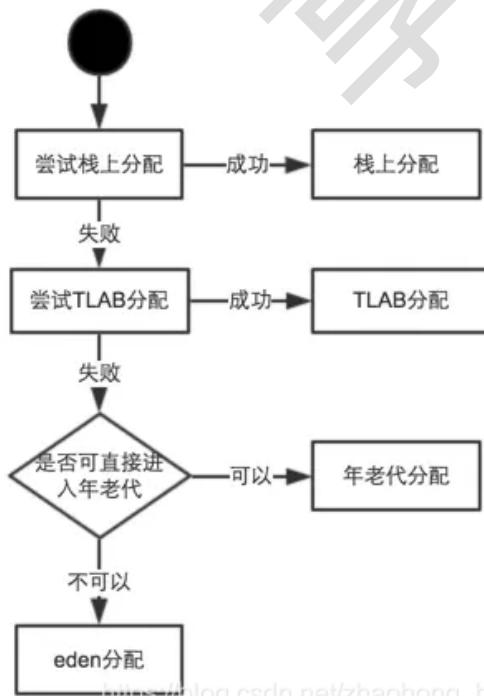
16.1.4Java堆内存的10个要点

1. Java堆内存是操作系统分配给JVM的内存的一部分。
2. 当我们创建对象时，它们存储在Java堆内存中。
3. 为了便于垃圾回收，Java堆空间分成三个区域，分别叫作New Generation, Old Generation或叫作Tenured Generation，还有Perm Space。
4. 你可以通过用JVM的命令行选项 -Xms, -Xmx, -Xmn来调整Java堆空间的大小。不要忘了在大小后面加上“M”或者“G”来表示单位。举个例子，你可以用 -Xmx256m来设置堆内存最大的大小为256MB。

5. 你可以用JConsole或者 Runtime.maxMemory(), Runtime.totalMemory(), Runtime.freeMemory()来查看Java中堆内存的大小。
6. 你可以使用命令“jmap”来获得heap dump，用“jhat”来分析heap dump。
7. Java堆空间不同于栈空间，栈空间是用来储存调用栈和局部变量的。
8. Java垃圾回收器是用来将死掉的对象(不再使用的对象)所占用的内存收回回来，再释放到Java堆空间中。
9. 当你遇到java.lang.OutOfMemoryError时，不要紧张，有时候仅仅增加堆空间就可以了，但如果经常出现的话，就要看看Java程序中是不是存在内存泄露了。
10. 请使用Profiler和Heap dump分析工具来查看Java堆空间，可以查看给每个对象分配了多少内存。

16.2 Java对象内存分配策略

一般认为，Java对象都是在堆上分配的，但也有一些特殊情况。Java对象内存分配策略：



在Java中，典型的对象不在堆上分配的情况有两种：TLAB (Thread Local

Allocation Buffer) 和栈上分配 (严格来说TLAB也是属于堆，只是在 TLAB比较特殊) 。

16.2.1 栈上分配

JVM在Server模式下的逃逸分析可以分析出某个对象是否永远只在某个方法、线程的范围内，并没有“逃逸”出这个范围，逃逸分析的一个结果就是对于某些未逃逸对象可以直接在栈上分配，由于该对象一定是局部的，所以栈上分配不会有问题是。在实际的应用程序，尤其是大型程序中反而发现实施逃逸分析可能出现效果不稳定的情况，或因分析过程耗时但却无法有效判别出非逃逸对象而导致性能（即时编译的收益）有所下降，所以在很长的一段时间里，即使是Server Compiler，也默认不开启逃逸分析，甚至在某些版本（如JDK 1.6 Update18）中还曾经短暂地完全禁止了这项优化。

16.2.2 TLAB分配

对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用CAS和失败重试的方式保证更新操作的原子性；另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）。

JVM在内存新生代Eden Space中开辟了一小块区域，由线程私有，称作 TLAB (Thread-local allocation buffer)，默认设定为占用Eden Space 的1%。在Java程序中很多对象都是小对象且用过即丢，它们不存在线程共享也适合被快速GC，所以对于小对象通常JVM会优先分配在TLAB上，并且TLAB上的分配由于是线程私有所以没有锁开销。因此在实践中分配多个小对象的效率通常比分配一个大对象的效率要高。

哪个线程要分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时，才需要同步锁定。虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。通常默认的TLAB区域大小是Eden区域的

1%，当然也可以手工进行调整，对应的JVM参数是-XX:TLABWasteTargetPercent。

16.2.3为什么不在堆上分配

我们知道堆是由所有线程共享的，既然如此那它就是竞争资源，对于竞争资源，必须采取必要的同步，所以当使用new关键字在堆上分配对象时，是需要锁的。既然有锁，就必定存在锁带来的开销，而且由于是对整个堆加锁，相对而言锁的粒度还是比较大的，影响效率。而无论是TLAB还是栈都是线程私有的，私有即避免了竞争。

所以对于某些特殊情况，可以采取避免在堆上分配对象的办法，以提高对象创建和销毁的效率。

16.2.4对象内存分配的两种方法

为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

1) 指针碰撞(Serial、ParNew等带Compact过程的收集器)

假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”(Bump the Pointer)。

2) 空闲列表(CMS这种基于Mark-Sweep算法的收集器)

如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”(Free List)。

选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。因此，在使用Serial、ParNew等带Compact过程的收集器时，系统采用的分配算法是指针碰撞，而使用CMS这种基于Mark-Sweep算法的收集器时，通常采用空闲列表。

第十七节 类加载机制/双亲委托

17.1 java类加载机制和类加载器以及双亲委派原则解析

关于类加载器和类加载机制，这个也算是老生常谈了，可能大家或多或少都有一些了解，但是如果真的要让你说出个一二三来，可能还是有点困难，所以今天也是花了一天时间，从头复习了一遍。

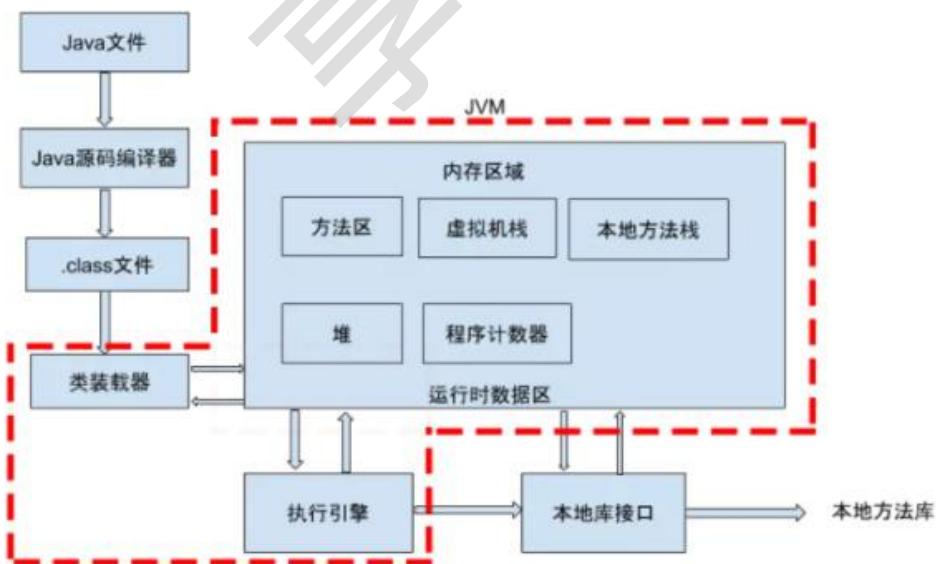
17.1.1类的加载是什么

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

只有java虚拟机才会创建Class对象，并且是一一对应关系，这样才能通过反射找到相应的类信息。

由下图我们可以看出类加载的过程在整个java程序运行期间处于一个怎样的环节：



从上图可以看，java文件通过编译器变成了.class文件，接下来类加载器又将这些.class文件加载到VM中。其中**类装载器**的作用其实就是类的加载。

类的加载由类加载器完成，类加载器通常由JVM提供，这些类加载器也是前面所有程序运行的基础，JVM提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承ClassLoader基类来创建自己的类加载器。

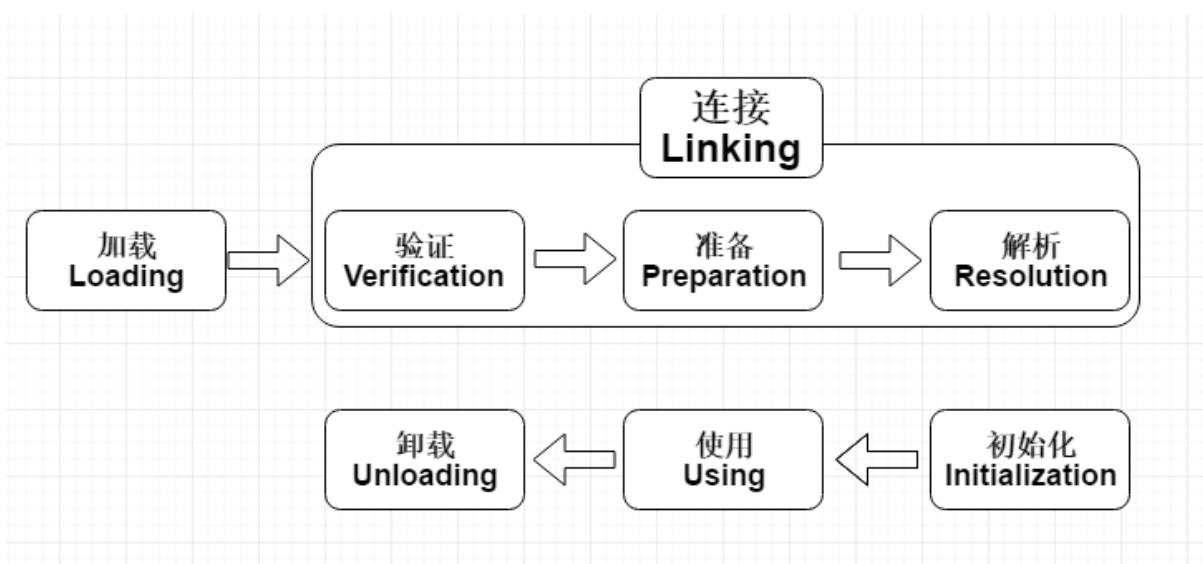
通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源。

- 从本地系统中直接加载
- 通过网络下载.class文件
- 从zip, jar等归档文件中加载.class文件
- 从专有数据库中提取.class文件
- 将Java源文件动态编译为.class文件

注意：类加载器通常无须等到“首次使用”该类时才加载该类，Java虚拟机规范允许系统预先加载某些类，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

17.1.2类加载器是什么

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。如下图：



其中类加载的过程包括了**加载、验证、准备、解析、初始化**五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

1、加载

"加载"是"类加载机制"的第一个过程，在加载阶段，虚拟机主要完成三件事：

- (1) 通过一个类的全限定名来获取其定义的二进制字节流
- (2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
- (3) 在堆中生成一个代表这个类的Class对象，作为方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段是可控性最强的阶段，因为程序员可以使用系统的类加载器加载，还可以使用自己的类加载器加载。我们在最后一部分会详细介绍这个类加载器。在这里我们只需要知道类加载器的作用就是上面虚拟机需要完成的三件事，仅此而已就好了。

2、验证

验证的主要作用就是确保被加载的类的正确性。也是连接阶段的第一步。说白了也就是我们加载好的.class文件不能对我们的虚拟机有危害，所以先检测验证一下。他主要是完成四个阶段的验证：

(1) 文件格式的验证：验证.class文件字节流是否符合class文件的格式的规范，并且能够被当前版本的虚拟机处理。这里面主要对魔数、主版本号、常量池等等的校验（魔数、主版本号都是.class文件里面包含的数据信息、在这里可以不用理解）。

(2) 元数据验证：主要是对字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范的要求，比如说验证这个类是不是有父类，类中的字段方法是不是和父类冲突等等。

(3) 字节码验证：这是整个验证过程最复杂的阶段，主要是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在元数据验证阶段对数据类型做出验证后，这个阶段主要对类的方法做出分析，保证类的方法在运行时不会做出威胁虚拟机安全的事。

(4) 符号引用验证：它是验证的最后一个阶段，发生在虚拟机将符号引用转化为直接引用的时候。主要是对类自身以外的信息进行校验。目的是确保解析动作能够完成。

对整个类加载机制而言，验证阶段是一个很重要但是非必需的阶段，如果我们的代码能够确保没有问题，那么我们就没有必要去验证，毕竟验证需要花费一定的时间。当然我们可以使用-Xverify:none来关闭大部分的验证。

3、准备

准备阶段主要为类变量分配内存并设置初始值。这些内存都在方法区分配。在这个阶段我们只需要注意两点就好了，也就是类变量和初始值两个关键词：

(1) 类变量 (static) 会分配内存，但是实例变量不会，实例变量主要随着对象的实例化一块分配到java堆中，

(2) 这里的初始值指的是数据类型默认值，而不是代码中被显示赋予的值。比如

```
public static int value = 1; //在这里准备阶段过后的value值为0，而不是1。赋值为1的动作在初始化阶段，当然还有其他的默认值。
```

注意，在上面value是被static所修饰的准备阶段之后是0，但是如果同时被final和static修饰准备阶段之后就是1了。我们可以理解为static final在编译器就将结果放入调用它的类的常量池中了。

4、解析

解析阶段主要是虚拟机将常量池中的符号引用转化为直接引用的过程。什么是符号应用和直接引用呢？

符号引用：以一组符号来描述所引用的目标，可以是任何形式的字面量，只要是能无歧义的定位到目标就好，就好比在班级中，老师可以用张三来代表你，也可以用你的学号来代表你，但无论任何方式这些都只是一个代号（符号），这个代号指向你（符号引用）直接引用：直接引用是可以指向目标的指针、相对偏移量或者是一个能直接或间接定位到目标的句柄。和虚拟机实现的内存有关，不同的虚拟机直接引用一般不同。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

5、初始化

这是类加载机制的最后一步，在这个阶段，java程序代码才开始真正执行。我们知道，在准备阶段已经为类变量赋过一次值。在初始化阶段，程序员可以根据自己的需求来赋值了。一句话描述这个阶段就是执行类构造器<clinit>()方法的过程。

在初始化阶段，主要为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化。在Java中对类变量进行初始值设定有两种方式：

- ①声明类变量是指定初始值
- ②使用静态代码块为类变量指定初始值

JVM初始化步骤

- 1、假如这个类还没有被加载和连接，则程序先加载并连接该类
- 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类
- 3、假如类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机

只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

- 创建类的实例，也就是new的方式访问某个类或接口的静态变量
- 或者对该静态变量赋值
- 调用类的静态方法

- 反射 (如 Class.forName("com.shengsiyuan.Test")) 初始化某个类的子类，则其父类也会被初始化
- Java虚拟机启动时被标明为启动类的类 (JavaTest)
- 直接使用 java.exe 命令来运行某个主类

17.1.3 类加载器是什么

类加载器的任务是根据一个类的全限定名来读取此类的二进制字节流到JVM中，然后转换为一个与目标类对应的java.lang.Class对象实例，在虚拟机提供了3种类加载器，引导 (Bootstrap) 类加载器、扩展 (Extension) 类加载器、系统 (System) 类加载器（也称应用类加载器），每种加载器的具体工作任务如下：

启动 (Bootstrap) 类加载器：引导类加载器是用本地代码实现的类加载器，它负责将 <JAVA_HOME>/lib下面的核心类库或 -Xbootclasspath选项指定的jar包等虚拟机识别的类库加载到内存中。由于启动类加载器是由native方法实现加载过程，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。

扩展 (Extension) 类加载器：扩展类加载器是由Sun的ExtClassLoader (sun.misc.Launcher\$ExtClassLoader) 实现的，它负责将 <JAVA_HOME>/lib/ext或者由系统变量-Djava.ext.dir指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。

系统 (System) 类加载器：系统类加载器是由 Sun 的AppClassLoader (sun.misc.Launcher\$AppClassLoader) 实现的，它负责将 用户类路径(java -classpath或-Djava.class.path变量所指的目录，即当前类所在路径及其引用的第三方类库的路径，如第四节中的问题6所述)下的类库加载到内存中。开发者可以直接使用系统类加载器。

但是：如果站在Java虚拟机的角度来讲，只存在两种不同的类加载器：

启动类加载器：它使用C++实现（这里仅限于Hotspot，也就是JDK1.5之后默认的虚拟机，有很多其他的虚拟机是用Java语言实现的），是虚拟机自身的一部分；

所有其他的类加载器：这些类加载器都由Java语言实现，独立于虚拟机之外，并且全部继承自抽象类java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

类加载器加载Class大致要经过如下8个步骤：

1、检测此Class是否载入过，即在缓冲区中是否有此Class，如果有直接进入第8步，否则进入第2步。

2、如果没有父类加载器，则要么Parent是根类加载器，要么本身就是根类加载器，则跳到第4步，如果父类加载器存在，则进入第3步。

3、请求使用父类加载器去载入目标类，如果载入成功则跳至第8步，否则接着执行第5步。

4、请求使用根类加载器去载入目标类，如果载入成功则跳至第8步，否则跳至第7步。

5、当前类加载器尝试寻找Class文件，如果找到则执行第6步，如果找不到则执行第7步。

6、从文件中载入Class，成功后跳至第8步。

7、抛出ClassNotFoundException异常。

8、返回对应的java.lang.Class对象。

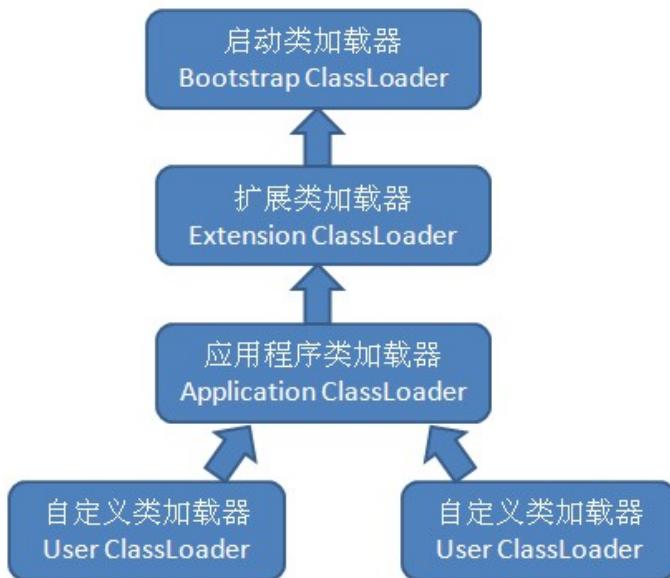
17.1.4 双亲委派机制

在Java中，采用双亲委派机制来实现类的加载。那什么是双亲委派机制？

所谓的双亲委派，则是先让父类加载器试图加载该Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父加载器，依次递归，如果父加载器可以完成类加载任务，就成功返回，只有父加载器无法完成此加载任务时，才自己去加载。

刚才我们说了，一个类加载器在加载一个类的时候，会让父类加载器

先去加载，而这个父类加载器是什么呢，如下图：



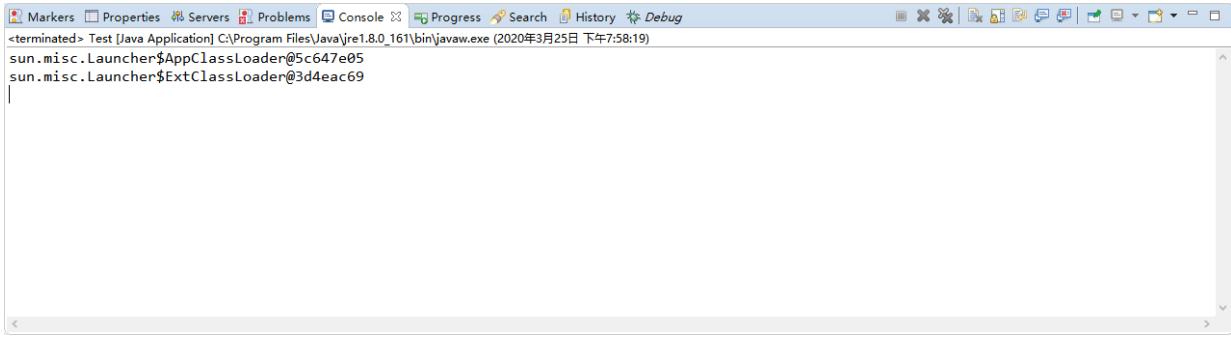
由上图我们可以看到类加载器之间的关系，但是他们的父子关系不是java中的继承，而是一种组合关系。

我们用以下代码来看看是不是这样的：

```
package main;

public class Test {
    public static void main(String[] args) {
        ClassLoader loader = Test.class.getClassLoader();
        while (loader != null) {
            System.out.println(loader);
            loader = loader.getParent();
        }
    }
}
```

运行结果如下所示：



从结果我们可以看出，默认情况下，用户自定义的类使用 AppClassLoader 加载，AppClassLoader 的父加载器为 ExtClassLoader，但是 ExtClassLoader 的父加载器却显示为空，这是什么原因呢？究其缘由，启动类加载器属于 JVM 的一部分，它不是由 Java 语言实现的，在 Java 中无法直接引用，所以才返回空。但如果是这样，该怎么实现 ExtClassLoader 与 启动类加载器之间双亲委派机制？我们可以参考一下源码：

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been
        loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c \= parent.loadClass(name,
false);
                } else {
                    c \=
findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if
                class not found
            }
        }
    }
}
```

```
// from the non-null parent class
loader
}

if (c == null) {
    // If still not found, then invoke
findClass in order
    // to find the class.
    long t1 = System.nanoTime();
    c \= findClass(name);

    // this is the defining class loader;
record the stats

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1
- t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTImeFro
m(t1);

sun.misc.PerfCounter.getFindClasses().increment();
}
}

if (resolve) {
    resolveClass(c);
}

return c;
}
}
```

从源码可以看出，ExtClassLoader 和 AppClassLoader都继承自 ClassLoader 类，ClassLoader 类中通过 loadClass 方法来实现双亲委派机制。整个类的加载过程可分为如下三步：

- 1、查找对应的类是否已经加载。

2、若未加载，则判断当前类加载器的父加载器是否为空，不为空则委托给父类去加载，否则调用启动类加载器加载
(findBootstrapClassOrNull 再往下会调用一个 native 方法)。

3、若第二步加载失败，则调用当前类加载器加载。

通过上面这段程序，可以很清楚的看出扩展类加载器与启动类加载器之间是如何实现委托模式的。

17.1.5 自定义类加载器

通常情况下，我们都是直接使用系统类加载器。但是，有的时候，我们也需要自定义类加载器。比如应用是通过网络来传输 Java 类的字节码，为保证安全性，这些字节码经过了加密处理，这时系统类加载器就无法对其进行加载，这样则需要自定义类加载器来实现。自定义类加载器一般都是继承自 ClassLoader 类，从上面对 loadClass 方法来分析来看，我们只需要重写 findClass 方法即可。下面我们通过一个示例来演示自定义类加载器的流程：

```
package main;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class NewClassLoader extends ClassLoader {

    private String root;

    protected class<?> findClass(String name) throws
ClassNotFoundException {
        byte[] classData = loadClassData(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
```

```
        return defineClass(name, classData, 0,
classData.length);
    }
}

private byte[] loadClassData(String className) {
    String fileName = root + File.separatorChar
        + className.replace('.', 
File.separatorChar) + ".class";
    try {
        InputStream ins = new
FileInputStream(fileName);
        ByteArrayOutputStream baos = new
ByteArrayOutputStream();
        int bufferSize = 1024;
        byte[] buffer = new byte[bufferSize];
        int length = 0;
        while ((length = ins.read(buffer)) != -1) {
            baos.write(buffer, 0, length);
        }
        return baos.toByteArray();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

public String getRoot() {
    return root;
}

public void setRoot(String root) {
    this.root = root;
}

public static void main(String[] args) {
```

```

        NewClassLoader classLoader \= new
NewClassLoader();

classLoader.setRoot("D:\\\\workplace\\\\tarzan\\\\Test\\\\
\\src\\\\");

    Class<?> testClass = null;
    try {
        //这里是完整的包路径
        testClass =
classLoader.loadClass("main.Test");
        Object object \= testClass.newInstance();

System.out.println(object.getClass().getClassLoader());
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
}

```

程序运行结果如下：

The screenshot shows a Java application running in an IDE. The code is identical to the one above, but the line `System.out.println(object.getClass().getClassLoader());` is highlighted in blue. The output window at the bottom shows the result:

```

123
main.NewClassLoader@5d22bbb7

```

The output "123" is the value printed by the `System.out.println(123);` statement. The string "main.NewClassLoader@5d22bbb7" is the memory address of the `NewClassLoader` object, which is the class loader used to load the `main.Test` class.

自定义类加载器的核心在于对字节码文件的获取，如果是加密的字节码则需要在该类中对文件进行解密。由于这里只是演示，我并未对class文件进行加密，因此没有解密的过程。这里有几点需要注意：

- 1、这里传递的文件名需要是类的全限定性名称，即 com.paddx.test.classloading.Test 格式的，因为 `defineClass` 方法是按这种格式进行处理的。
- 2、最好不要重写 `loadClass` 方法，因为这样容易破坏双亲委托模式。
- 3、这类 Test 类本身可以被 `AppClassLoader` 类加载，因此我们不能把 `com/paddx/test/classloading/Test.class` 放在类路径下。否则，由于双亲委托机制的存在，会直接导致该类由 `AppClassLoader` 加载，而不会通过我们自定义类加载器来加载。

好了，关于类加载就讲这么多了，有不对的地方还请各位指出。

第二章 Android基础

第一节 Activity知识点(必问)

1.1 Activity启动过程全解析

前言

- 一个App是怎么启动起来的？
- App的程序入口到底是哪里？
- Launcher到底是什么神奇的东西？
- 听说还有个AMS的东西，它是做什么的？
- Binder是什么？他是如何进行IPC通信的？
- Activity生命周期到底是什么时候调用的？被谁调用的？
- 等等...

你是不是还有很多类似的疑问一直没有解决？没关系，这篇文章将结合源码以及大量的优秀文章，站在巨人的肩膀上，更加通俗的来试着解释一些问题。但是毕竟源码繁多、经验有限，文中不免会出现一些纰漏甚至是错误，还恳请大家指出，互相学习。

学习目标

1. 了解从手机开机第一个zygote进程创建，到点击桌面上的图标，进入一个App的完整流程，并且从源码的角度了解到一个Activity的生命周期是怎么回事
2. 了解到ActivityManagerServices(即AMS)、ActivityStack、ActivityThread、Instrumentation等Android framework中非常重要的基础类的作用，及相互间的关系
3. 了解AMS与ActivityThread之间利用Binder进行IPC通信的过程，了解AMS和ActivityThread在控制Activity生命周期起到的作用和相互之间的配合
4. 了解与Activity相关的framework层的其他琐碎问题

写作方式

这篇文章我决定采用一问一答的方式进行。

其实在这之前，我试过把每个流程的代码调用过程，用粘贴源代码的方式写在文章里，但是写完一部分之后，发现由于代码量太大，整篇文章和老太太的裹脚布一样——又臭又长，虽然每个重要的操作可以显示出详细调用过程，但是太关注于细节反而导致从整体上不能很好的把握。所以在原来的基础之上进行了修改，对关键的几个步骤进行重点介绍，力求语言简洁，重点突出，从而让大家在更高的层次上对framework层有个认识，然后结合后面我给出的参考资料，大家就可以更加快速，更加高效的了解这一块的整体架构。

主要对象功能介绍

我们下面的文章将围绕着这几个类进行介绍。可能你第一次看的时候，印象不深，不过没关系，当你跟随者我读完这篇文章的时候，我会在最后再次列出这些对象的功能，相信那时候你会对这些类更加的熟悉和深刻。

- ActivityManagerServices，简称AMS，服务端对象，负责系统中所有Activity的生命周期
- ActivityThread，App的真正入口。当开启App之后，会调用main()开始运行，开启消息循环队列，这就是传说中的UI线程或者叫主线程。与ActivityManagerServices配合，一起完成Activity的管理工作

- ApplicationThread，用来实现ActivityManagerService与ActivityThread之间的交互。在ActivityManagerService需要管理相关Application中的Activity的生命周期时，通过ApplicationThread的代理对象与ActivityThread通讯。
- ApplicationThreadProxy，是ApplicationThread在服务器端的代理，负责和客户端的ApplicationThread通讯。AMS就是通过该代理与ActivityThread进行通信的。
- Instrumentation，每一个应用程序只有一个Instrumentation对象，每个Activity内都有一个对该对象的引用。Instrumentation可以理解为应用进程的管家，ActivityThread要创建或暂停某个Activity时，都需要通过Instrumentation来进行具体的操作。
- ActivityStack，Activity在AMS的栈管理，用来记录已经启动的Activity的先后关系，状态信息等。通过ActivityStack决定是否需要启动新的进程。
- ActivityRecord，ActivityStack的管理对象，每个Activity在AMS对应一个ActivityRecord，来记录Activity的状态以及其他管理信息。其实质就是服务器端的Activity对象的映像。
- TaskRecord，AMS抽象出来的一个“任务”的概念，是记录ActivityRecord的栈，一个“Task”包含若干个ActivityRecord。AMS用TaskRecord确保Activity启动和退出的顺序。如果你清楚Activity的4种launchMode，那么对这个概念应该不陌生。

主要流程介绍

下面将按照App启动过程的先后顺序，一问一答，来解释一些事情。

让我们开始吧！

zygote是什么？有什么作用？

首先，你觉得这个单词眼熟不？当你的程序Crash的时候，打印的红色log下面通常带有这一个单词。

zygote意为“受精卵”。Android是基于Linux系统的，而在Linux中，所有的进程都是由init进程直接或者是间接fork出来的，zygote进程也不例外。

在Android系统里面，zygote是一个进程的名字。Android是基于Linux System的，当你的手机开机的时候，Linux的内核加载完成之后就会启动一个叫“init”的进程。在Linux System里面，所有的进程都是由init进程fork出来的，我们的zygote进程也不例外。

我们都知道，每一个App其实都是

- 一个单独的dalvik虚拟机
- 一个单独的进程

所以当系统里面的第一个zygote进程运行之后，在这之后再开启App，就相当于开启一个新的进程。而为了实现资源共用和更快的启动速度，Android系统开启新进程的方式，是通过fork第一个zygote进程实现的。所以说，除了第一个zygote进程，其他应用所在的进程都是zygote的子进程，这下你明白为什么这个进程叫“受精卵”了吧？因为就像是一个受精卵一样，它能快速的分裂，并且产生遗传物质一样的细胞！

SystemServer是什么？有什么作用？它与zygote的关系是什么？

首先我要告诉你是的，SystemServer也是一个进程，而且是由zygote进程fork出来的。

知道了SystemServer的本质，我们对它就不算太陌生了，这个进程是Android Framework里面两大非常重要的进程之一——另外一个进程就是上面的zygote进程。

为什么说SystemServer非常重要呢？因为系统里面重要的服务都是在这个进程里面开启的，比如

ActivityManagerService、PackageManagerService、WindowManagerService等等，看着是不是都挺眼熟的？

那么这些系统服务是怎么开启起来的呢？

在zygote开启的时候，会调用ZygoteInit.main()进行初始化

```
public static void main(String argv[]) {  
    ...ignore some code...  
}
```

```

//在加载首个zygote的时候，会传入初始化参数，使得
startSystemServer = true
    boolean startSystemServer = false;
    for (int i = 1; i < argv.length; i++) {
        if ("start-system-
server".equals(argv[i])) {
            startSystemServer = true;
        } else if
(argv[i].startsWith(ABI_LIST_ARG)) {
            abiList =
argv[i].substring(ABI_LIST_ARG.length());
        } else if
(argv[i].startsWith(SOCKET_NAME_ARG)) {
            socketName =
argv[i].substring(SOCKET_NAME_ARG.length());
        } else {
            throw new RuntimeException("Unknown
command line argument: " + argv[i]);
        }
    }

    ...ignore some code...

    //开始fork我们的SystemServer进程
    if (startSystemServer) {
        startSystemServer(abiList, socketName);
    }

    ...ignore some code...
}

}

```

我们看下startSystemServer()做了些什么

```

/**留着这个注释，就是为了说明SystemServer确实是被fork出来的
 * Prepare the arguments and fork for the system
server process.
 */

```

```
    private static boolean startSystemServer(String
abiList, String socketName)
        throws MethodAndArgsCaller, RuntimeException
{

    ...ignore some code...

    //留着这段注释，就是为了说明上面ZygoteInit.main(String
    argv[])
    /* Hardcoded command line to start the system
server */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
        "--
setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,10
10,1018,1032,3001,3002,3003,3006,3007",
        "--capabilities=" + capabilities + "," +
capabilities,
        "--runtime-init",
        "--nice-name=system_server",
        "com.android.server.SystemServer",
    };
    int pid;
    try {
        parsedArgs = new
ZygoteConnection.Arguments(args);

        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);

        ZygoteConnection.applyInvokewithSystemProperty(parsedArgs
);
    }

    //确实是fuck出来的吧，我没骗你吧~不对，是fork出来的 -_-
|||
    /* Request to fork the system server process
*/
```

```
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }

        /* For child process */
        if (pid == 0) {
            if (hasSecondZygote(abiList)) {
                waitForSecondaryZygote(socketName);
            }

            handleSystemServerProcess(parsedArgs);
        }

        return true;
    }
}
```

ActivityManagerService是什么？什么时候初始化的？有什么作用？

ActivityManagerService，简称AMS，服务端对象，负责系统中所有Activity的生命周期。

ActivityManagerService进行初始化的时机很明确，就是在SystemServer进程开启的时候，就会初始化ActivityManagerService。从下面的代码中可以看到

```
public final class SystemServer {

    //zygote的主入口
    public static void main(String[] args) {
        new SystemServer().run();
    }
}
```

```
public SystemServer() {
    // Check for factory test mode.
    mFactoryTestMode = FactoryTest.getMode();
}

private void run() {
    ...ignore some code...

    //加载本地系统服务库，并进行初始化
    System.loadLibrary("android_servers");
    nativeInit();

    // 创建系统上下文
    createSystemContext();

    //初始化SystemServiceManager对象，下面的系统服务开启都需要调用SystemServiceManager.startService(Class<T>)，这个方法通过反射来启动对应的服务
    mSystemServiceManager = new
    SystemServiceManager(mSystemContext);

    //开启服务
    try {
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    } catch (Throwable ex) {
        slog.e("System",
"*****");
        slog.e("System", "***** Failure
starting system services", ex);
        throw ex;
    }

    ...ignore some code...
}
```

```
//初始化系统上下文对象mSystemContext，并设置默认的主题，mSystemContext实际上是一个ContextImpl对象。调用ActivityThread.systemMain()的时候，会调用ActivityThread.attach(true)，而在attach()里面，则创建了Application对象，并调用了Application.onCreate()。  
private void createSystemContext() {  
    ActivityThread activityThread =  
ActivityThread.systemMain();  
    mSystemContext =  
activityThread.getSystemContext();  
  
mSystemContext.setTheme(android.R.style.Theme_DeviceDefault_Light_DarkActionBar);  
}  
  
//在这里开启了几个核心的服务，因为这些服务之间相互依赖，所以都放在了这个方法里面。  
private void startBootstrapServices() {  
  
    ...ignore some code...  
  
    //初始化ActivityManagerService  
    mActivityManagerService =  
mSystemServiceManager.startService(  
ActivityManagerService.Lifecycle.class).getService();  
  
mActivityManagerService.setSystemServiceManager(mSystemServiceManager);  
  
    //初始化PowerManagerService，因为其他服务需要依赖这个Service，因此需要尽快的初始化  
    mPowerManagerService =  
mSystemServiceManager.startService(PowerManagerService.class);
```

```
// 现在电源管理已经开启，ActivityManagerService负责电
源管理功能
    mActivityManagerService.initPowerManagement();

    // 初始化DisplayManagerService
    mDisplayManagerService =
mSystemServiceManager.startService(DisplayManagerService.
class);

    //初始化PackageManagerService
    PackageManagerService =
PackageManagerService.main(mSystemContext, mInstaller,
        mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
        mOnlyCore);

    ...ignore some code...

}

}
```

经过上面这些步骤，我们的ActivityManagerService对象已经创建好了，并且完成了成员变量初始化。而且在这之前，调用createSystemContext()创建系统上下文的时候，也已经完成了mSystemContext和ActivityThread的创建。注意，这是系统进程开启时的流程，在这之后，会开启系统的Launcher程序，完成系统界面的加载与显示。

你是否会好奇，我为什么说AMS是服务端对象？下面我给你介绍下Android系统里面的服务器和客户端的概念。

其实服务器客户端的概念不仅仅存在于Web开发中，在Android的框架设计中，使用的也是这一种模式。服务器端指的就是所有App共用的系统服务，比如我们这里提到的ActivityManagerService，和前面提到的PackageManagerService、WindowManagerService等等，这些基础的系统服务是被所有的App公用的，当某个App想实现某个操作的时候，要

告诉这些系统服务，比如你想打开一个App，那么我们知道了包名和MainActivity类名之后就可以打开

```
Intent intent = new Intent(Intent.ACTION_MAIN);
intent.addCategory(Intent.CATEGORY_LAUNCHER);

ComponentName cn = new ComponentName(packageName,
className);
intent.setComponent(cn);
startActivity(intent);
```

但是，我们的App通过调用startActivity()并不能直接打开另外一个App，这个方法会通过一系列的调用，最后还是告诉AMS说：“我要打开这个App，我知道他的住址和名字，你帮我打开吧！”所以是AMS来通知zygote进程来fork一个新进程，来开启我们的目标App的。这就像是浏览器想要打开一个超链接一样，浏览器把网页地址发送给服务器，然后还是服务器把需要的资源文件发送给客户端的。

知道了Android Framework的客户端服务器架构之后，我们还需要了解一件事情，那就是我们的App和AMS(SystemServer进程)还有zygote进程分属于三个独立的进程，他们之间如何通信呢？

App与AMS通过Binder进行IPC通信，AMS(SystemServer进程)与zygote通过Socket进行IPC通信。

那么AMS有什么用呢？在前面我们知道了，如果想打开一个App的话，需要AMS去通知zygote进程，除此之外，其实所有的Activity的开启、暂停、关闭都需要AMS来控制，所以我们说，AMS负责系统中所有Activity的生命周期。

在Android系统中，任何一个Activity的启动都是由AMS和应用程序进程（主要是ActivityThread）相互配合来完成的。AMS服务统一调度系统中所有进程的Activity启动，而每个Activity的启动过程则由其所属的进程具体来完成。

这样说你可能还是觉得比较抽象，没关系，下面有一部分是专门来介绍AMS与ActivityThread如何一起合作控制Activity的生命周期的。

Launcher是什么？什么时候启动的？

当我们点击手机桌面上的图标的时候，App就由Launcher开始启动了。但是，你有没有思考过Launcher到底是一个什么东西？

Launcher本质上也是一个应用程序，和我们的App一样，也是继承自Activity

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
public final class Launcher extends Activity
    implements View.OnClickListener,
OnLongClickListener, LauncherModel.Callbacks,
    View.OnTouchListener {
}
```

Launcher实现了点击、长按等回调接口，来接收用户的输入。既然是普通的App，那么我们的开发经验在这里就仍然适用，比如，我们点击图标的时候，是怎么开启的应用呢？如果让你，你怎么做这个功能呢？捕捉图标点击事件，然后startActivity()发送对应的Intent请求呗！是的，Launcher也是这么做的，就是这么easy！

那么到底是处理的哪个对象的点击事件呢？既然Launcher是App，并且有界面，那么肯定有布局文件呀，是的，我找到了布局文件launcher.xml

```
<FrameLayout

    xmlns:android="http://schemas.android.com/apk/res/android"
    "


    xmlns:launcher="http://schemas.android.com/apk/res/com.android.launcher"
    android:id="@+id/launcher">

    <com.android.launcher2.DragLayer
        android:id="@+id/drag_layer"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fitsSystemWindows="true">
```

```
<!-- Keep these behind the workspace so that they  
are not visible when  
    we go into AllApps -->  
<include  
    android:id="@+id/dock_divider"  
    layout="@layout/workspace_divider"  
  
    android:layout_marginBottom="@dimen/button_bar_height"  
    android:layout_gravity="bottom" />  
  
<include  
    android:id="@+id/paged_view_indicator"  
    layout="@layout/scroll_indicator"  
    android:layout_gravity="bottom"  
  
    android:layout_marginBottom="@dimen/button_bar_height" />  
  
<!-- The workspace contains 5 screens of cells -->  
<com.android.launcher2.workspace  
    android:id="@+id/workspace"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
  
    android:paddingStart="@dimen/workspace_left_padding"  
  
    android:paddingEnd="@dimen/workspace_right_padding"  
  
    android:paddingTop="@dimen/workspace_top_padding"  
  
    android:paddingBottom="@dimen/workspace_bottom_padding"  
    launcher:defaultScreen="2"  
    launcher:cellCountX="@integer/cell_count_x"  
    launcher:cellCountY="@integer/cell_count_y"  
  
    launcher:pageSpacing="@dimen/workspace_page_spacing"
```

```
launcher:scrollIndicatorPaddingLeft="@dimen/workspace_divider_padding_left"

launcher:scrollIndicatorPaddingRight="@dimen/workspace_divider_padding_right">

    <include android:id="@+id/cell1"
layout="@layout/workspace_screen" />
    <include android:id="@+id/cell2"
layout="@layout/workspace_screen" />
    <include android:id="@+id/cell3"
layout="@layout/workspace_screen" />
    <include android:id="@+id/cell4"
layout="@layout/workspace_screen" />
    <include android:id="@+id/cell5"
layout="@layout/workspace_screen" />
</com.android.launcher2.Workspace>

    ...ignore some code...

    </com.android.launcher2.DragLayer>
</FrameLayout>
```

为了方便查看，我删除了很多代码，从上面这些我们应该可以看出一些东西来：Launcher大量使用标签来实现界面的复用，而且定义了很多的自定义控件实现界面效果，dock_divider从布局的参数声明上可以猜出，是底部操作栏和上面图标布局的分割线，而paged_view_indicator则是页面指示器，和App首次进入的引导页下面的界面引导是一样的道理。当然，我们最关心的是Workspace这个布局，因为注释里面说在这里面包含了5个屏幕的单元格，想必你也猜到了，这个就是在首页存放我们图标的那五个界面(不同的ROM会做不同的DIY，数量不固定)。

接下来，我们应该打开workspace_screen布局，看看里面有什么东东。

workspace_screen.xml

```
<com.android.launcher2.CellLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
"

xmlns:launcher="http://schemas.android.com/apk/res/com.android.launcher"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"

    android:paddingStart="@dimen/cell_layout_left_padding"
    android:paddingEnd="@dimen/cell_layout_right_padding"
    android:paddingTop="@dimen/cell_layout_top_padding"

    android:paddingBottom="@dimen/cell_layout_bottom_padding"
    android:hapticFeedbackEnabled="false"
    launcher:cellWidth="@dimen/workspace_cell_width"
    launcher:cellHeight="@dimen/workspace_cell_height"
    launcher:widthGap="@dimen/workspace_width_gap"
    launcher:heightGap="@dimen/workspace_height_gap"
    launcher:maxGap="@dimen/workspace_max_gap" />
```

里面就一个CellLayout，也是一个自定义布局，那么我们就可以猜到了，既然可以存放图标，那么这个自定义的布局很有可能是继承自ViewGroup或者是其子类，实际上，CellLayout确实是继承自ViewGroup。在CellLayout里面，只放了一个子View，那就是ShortcutAndWidgetContainer。从名字也可以看出来，ShortcutAndWidgetContainer这个类就是用来存放**快捷图标**和**Widget小部件**的，那么里面放的是什么对象呢？

在桌面上的图标，使用的是BubbleTextView对象，这个对象在TextView的基础之上，添加了一些特效，比如你长按移动图标的时候，图标位置会出现一个背景(不同版本的效果不同)，所以我们找到BubbleTextView对象的点击事件，就可以找到Launcher如何开启一个App了。

除了在桌面上有图标之外，在程序列表中点击图标，也可以开启对应的程序。这里的图标使用的不是BubbleTextView对象，而是PagedViewIcon对象，我们如果找到它的点击事件，就也可以找到Launcher如何开启一个App。

其实说这么多，和今天的主题隔着十万八千里，上面这些东西，你有兴趣就看，没兴趣就直接跳过，不知道不影响这篇文章阅读。

BubbleTextView的点击事件在哪里呢？我来告诉你：在 Launcher.onClick(View v)里面。

```
/**
 * Launches the intent referred by the clicked
shortcut
*/
public void onClick(View v) {

    ...ignore some code...

    Object tag = v.getTag();
    if (tag instanceof ShortcutInfo) {
        // Open shortcut
        final Intent intent = ((ShortcutInfo)
tag).intent;
        int[] pos = new int[2];
        v.getLocationOnScreen(pos);
        intent.setSourceBounds(new Rect(pos[0],
pos[1],
pos[0] + v.getWidth(), pos[1] +
v.getHeight()));
        //开始开启Activity咯~
        boolean success = startActivitySafely(v,
intent, tag);

        if (success && v instanceof BubbleTextView) {
            mWaitingForResume = (BubbleTextView) v;
            mWaitingForResume.setStayPressed(true);
        }
    } else if (tag instanceof FolderInfo) {
        //如果点击的是图标文件夹，就打开文件夹
        if (v instanceof FolderIcon) {
            FolderIcon fi = (FolderIcon) v;
            handleFolderClick(fi);
        }
    }
}
```

```
        }

    } else if (v == mAllAppsButton) {
        ...ignore some code...
    }

}
```

从上面的代码我们可以看到，在桌面上点击快捷图标的时候，会调用

```
startActivitySafely(v, intent, tag);
```

那么从程序列表界面，点击图标的时候会发生什么呢？实际上，程序列表界面使用的是AppsCustomizePagedView对象，所以我在类里面找到了onClick(View v)。

com.android.launcher2.AppsCustomizePagedView.java

```
/***
 * The Apps/Customize page that displays all the
applications, widgets, and shortcuts.
 */

public class AppsCustomizePagedView extends
PagedViewwithDraggableItems implements
    View.OnClickListener, View.OnKeyListener,
DragSource,
    PagedViewIcon.PressedCallback,
Pagedviewwidget.ShortPressListener,
    LauncherTransitionable {

    @Override
    public void onClick(View v) {

        ...ignore some code...

        if (v instanceof PagedViewIcon) {
            mLauncher.updatewallpaperVisibility(true);
            mLauncher.startActivitySafely(v,
appInfo.intent, appInfo);
        } else if (v instanceof Pagedviewwidget) {
            ...ignore some code..
    }
}
```

```
    }  
}  
}
```

可以看到，调用的是

```
mLauncher.startActivitySafely(v, appInfo.intent,  
appInfo);
```

和上面一样！这叫什么？这叫殊途同归！

所以咱们现在又明白了一件事情：不管从哪里点击图标，调用的都是 Launcher.startActivitySafely()。

- 下面我们就可以一步步的来看一下Launcher.startActivitySafely()到底做了什么事情。

```
boolean startActivitySafely(View v, Intent intent,  
Object tag) {  
    boolean success = false;  
    try {  
        success = startActivity(v, intent, tag);  
    } catch (ActivityNotFoundException e) {  
        Toast.makeText(this,  
R.string.activity_not_found, Toast.LENGTH_SHORT).show();  
        Log.e(TAG, "Unable to launch. tag=" + tag + "  
intent=" + intent, e);  
    }  
    return success;  
}
```

调用了startActivity(v, intent, tag)

```
boolean startActivity(View v, Intent intent, Object tag)  
{  
  
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
    try {  
        boolean useLaunchAnimation = (v != null) &&
```

```
!intent.hasExtra(INTENT_EXTRA_IGNORE_LAUNCH_ANIMATION);

        if (useLaunchAnimation) {
            if (user == null ||
user.equals(android.os.Process.myUserHandle())) {
                startActivity(intent,
opts.toBundle());
            } else {
                launcherApps.startActivity(intent.getComponent(),
user,
                    intent.getSourceBounds(),
opts.toBundle());
            }
        } else {
            if (user == null ||
user.equals(android.os.Process.myUserHandle())) {
                startActivity(intent);
            } else {
                launcherApps.startActivity(intent.getComponent(),
user,
                    intent.getSourceBounds(),
null);
            }
        }
        return true;
    } catch (SecurityException e) {
        ...
    }
    return false;
}
```

这里会调用Activity.startActivity(intent, opts.toBundle()), 这个方法熟悉吗？这就是我们经常用到的Activity.startActivity(Intent)的重载函数。而且由于设置了

```
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

所以这个Activity会添加到一个新的Task栈中，而且，`startActivity()`调用的其实是`startActivityForResult()`这个方法。

```
@Override  
    public void startActivityForResult(Intent intent, @Nullable  
Bundle options) {  
    if (options != null) {  
        startActivityForResult(intent, -1, options);  
    } else {  
        // Note we want to go through this call for  
        // compatibility with  
        // applications that may have overridden the  
        // method.  
        startActivityForResult(intent, -1);  
    }  
}
```

所以我们现在明确了，Launcher中开启一个App，其实和我们在Activity中直接`startActivity()`基本一样，都是调用了`Activity.startActivityForResult()`。

Instrumentation是什么？和ActivityThread是什么关系？

还记得前面说过的Instrumentation对象吗？每个Activity都持有Instrumentation对象的一个引用，但是整个进程只会存在一个Instrumentation对象。当`startActivityForResult()`调用之后，实际上还是调用了`mInstrumentation.execStartActivity()`

```
public void startActivityForResult(Intent intent, int  
requestCode, @Nullable Bundle options) {  
    if (mParent == null) {  
        Instrumentation.ActivityResult ar =  
            mInstrumentation.execStartActivity(  
                this,  
                mMainThread.getApplicationThread(), mToken, this,  
                intent, requestCode, options);  
    }  
}
```

```
        if (ar != null) {
            mMainThread.sendActivityResult(
                mToken, mEmbeddedID, requestCode,
                ar.getResultCode(),
                ar.getResultData());
        }
        ...ignore some code...
    } else {
        if (options != null) {
            //当现在的Activity有父Activity的时候会调用,
           但是在startActivityFromChild()内部实际还是调用的
            mInstrumentation.execStartActivity()
                mParent.startActivityFromChild(this,
                intent, requestCode, options);
        } else {
            mParent.startActivityFromChild(this,
                intent, requestCode);
        }
        ...ignore some code...
    }
}
```

下面是mInstrumentation.execStartActivity()的实现

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder
    token, Activity target,
    Intent intent, int requestCode, Bundle
    options) {
    IApplicationThread whoThread =
    (IApplicationThread) contextThread;
    ...ignore some code...
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess();
        int result =
        ActivityManagerNative.getDefault()
```

```
        .startActivity(whoThread,
who.getPackageName(), intent,
intent.resolveTypeIfNeeded(who.getContentResolver()),
token, target != null ?
target.mEmbeddedID : null,
requestCode, 0, null, options);
checkStartActivityResult(result, intent);
} catch (RemoteException e) {
}
return null;
}
```

所以当我们在程序中调用startActivity()的时候，实际上调用的是Instrumentation的相关的方法。

Instrumentation意为“仪器”，我们先看一下这个类里面包含哪些方法吧

我们可以看到，这个类里面的方法大多数和Application和Activity有关，是的，这个类就是完成对Application和Activity初始化和生命周期的工具类。比如说，我单独挑一个callActivityOnCreate()让你看看

```
public void callActivityOnCreate(Activity activity,
Bundle icicle) {
    prePerformCreate(activity);
    activity.performCreate(icicle);
    postPerformCreate(activity);
}
```

对activity.performCreate(icicle);这一行代码熟悉吗？这一行里面就调用了传说中的Activity的入口函数onCreate()，不信？接着往下看

Activity.performCreate()

```
final void performCreate(Bundle icicle) {  
    onCreate(icicle);  
    mActivityTransitionState.readState(icicle);  
    performCreateCommon();  
}
```

没骗你吧，onCreate在这里调用了吧。但是有一件事情必须说清楚，那就是这个Instrumentation类这么重要，为啥我在开发的过程中，没有发现他的踪迹呢？

是的，Instrumentation这个类很重要，对Activity生命周期方法的调用根本就离不开他，他可以说是一个大管家，但是，这个大管家比较害羞，是一个女的，管内不管外，是老板娘~

那么你可能要问了，老板是谁呀？

老板当然是大名鼎鼎的ActivityThread了！

ActivityThread你都没听说过？那你肯定听说过传说中的UI线程吧？是的，这就是UI线程。我们前面说过，App和AMS是通过Binder传递信息的，那么ActivityThread就是专门与AMS的外交工作的。

AMS说：“ActivityThread，你给我暂停一个Activity！”

ActivityThread就说：“没问题！”然后转身和Instrumentation说：“老婆，AMS让暂停一个Activity，我这里忙着呢，你快去帮我把这事办了把~”

于是，Instrumentation就去把事儿搞定了。

所以说，AMS是董事会，负责指挥和调度的，ActivityThread是老板，虽然说家里的事自己说了算，但是需要听从AMS的指挥，而Instrumentation则是老板娘，负责家里的大事小事，但是一般不抛头露面，听一家之主ActivityThread的安排。

如何理解AMS和ActivityThread之间的Binder通信？

前面我们说到，在调用startActivity()的时候，实际上调用的是

```
mInstrumentation.execStartActivity()
```

但是到这里还没完呢！里面又调用了下面的方法

```
ActivityManagerNative.getDefault()  
    .startActivity
```

这里的ActivityManagerNative.getDefault返回的就是ActivityManagerService的远程接口，即ActivityManagerProxy。

怎么知道的呢？往下看

```
public abstract class ActivityManagerNative extends  
Binder implements IActivityManager  
{  
  
    //从类声明上，我们可以看到ActivityManagerNative是Binder的一个子  
    //类，而且实现了IActivityManager接口  
    static public IActivityManager getDefault() {  

```

```
//最终返回的还是一个ActivityManagerProxy对象
static public IActivityManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }
    IActivityManager in =
(IActivityManager)obj.queryLocalInterface(descriptor);
    if (in != null) {
        return in;
    }

    //这里面的Binder类型的obj参数会作为ActivityManagerProxy的
    //成员变量保存为mRemote成员变量，负责进行IPC通信
    return new ActivityManagerProxy(obj);
}

}
```

再看ActivityManagerProxy.startActivity()，在这里面做的事情就是IPC通信，利用Binder对象，调用transact()，把所有需要的参数封装成Parcel对象，向AMS发送数据进行通信。

```
public int startActivity(IApplicationThread caller,
String callingPackage, Intent intent,
        String resolvedType, IBinder resultTo, String
resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo,
Bundle options) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();

    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ?
caller.asBinder() : null);
    data.writeString(callingPackage);
```

```
        intent.writeToParcel(data, 0);
        data.writeString(resolvedType);
        data.writeStrongBinder(resultTo);
        data.writeString(resultWho);
        data.writeInt(requestCode);
        data.writeInt(startFlags);
        if (profilerInfo != null) {
            data.writeInt(1);
            profilerInfo.writeToParcel(data,
Parcelables.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            data.writeInt(0);
        }
        if (options != null) {
            data.writeInt(1);
            options.writeToParcel(data, 0);
        } else {
            data.writeInt(0);
        }
        mRemote.transact(START_ACTIVITY_TRANSACTION,
data, reply, 0);
        reply.readException();
        int result = reply.readInt();
        reply.recycle();
        data.recycle();
        return result;
    }
}
```

Binder本质上只是一种底层通信方式，和具体服务没有关系。为了提供具体服务，Server必须提供一套接口函数以便Client通过远程访问使用各种服务。这时通常采用Proxy设计模式：将接口函数定义在一个抽象类中，Server和Client都会以该抽象类为基类实现所有接口函数，所不同的是Server端是真正的功能实现，而Client端是对这些函数远程调用请求的包装。

为了更方便的说明客户端和服务器之间的Binder通信，下面以ActivityManagerServices和他在客户端的代理类ActivityManagerProxy为例。

ActivityManagerServices和ActivityManagerProxy都实现了同一个接口——IActivityManager。

```
class ActivityManagerProxy implements IActivityManager{  
  
    public final class ActivityManagerService extends  
        ActivityManagerNative{}  
  
    public abstract class ActivityManagerNative extends  
        Binder implements IActivityManager{}
```

虽然都实现了同一个接口，但是代理对象ActivityManagerProxy并不会对这些方法进行真正地实现，ActivityManagerProxy只是通过这种方式对方方法的参数进行打包(因为都实现了相同接口，所以可以保证同一个方法有相同的参数，即对要传输给服务器的数据进行打包)，真正实现的是ActivityManagerService。

但是这个地方并不是直接由客户端传递给服务器，而是通过Binder驱动进行中转。其实我对Binder驱动并不熟悉，我们就把他当做一个中转站就OK，客户端调用ActivityManagerProxy接口里面的方法，把数据传送给Binder驱动，然后Binder驱动就会把这些东西转发给服务器的ActivityManagerServices，由ActivityManagerServices去真正的实施具体的操作。

但是Binder只能传递数据，并不知道是要调用ActivityManagerServices的哪个方法，所以在数据中会添加方法的唯一标识码，比如前面的startActivity()方法：

```
public int startActivity(IApplicationThread caller,  
    String callingPackage, Intent intent,  
    String resolvedType, IBinder resultTo, String  
    resultWho, int requestCode,  
    int startFlags, ProfilerInfo profilerInfo,  
    Bundle options) throws RemoteException {  
    Parcel data = Parcel.obtain();  
    Parcel reply = Parcel.obtain();
```

```
...ignore some code...
```

```
mRemote.transact(START_ACTIVITY_TRANSACTION,  
data, reply, 0);  
    reply.readException();  
    int result = reply.readInt();  
    reply.recycle();  
    data.recycle();  
    return result;  
}
```

上面的START_ACTIVITY_TRANSACTION就是方法标示， data是要传输给Binder驱动的数据， reply则接受操作的返回值。

即

客户端： ActivityManagerProxy =====> Binder驱动=====>
ActivityManagerService： 服务器

而且由于继承了同样的公共接口类， ActivityManagerProxy提供了与ActivityManagerService一样的函数原型，使用户感觉不出Server是运行在本地还是远端，从而可以更加方便的调用这些重要的系统服务。

但是！ 这里Binder通信是单方向的，即从ActivityManagerProxy指向ActivityManagerService的，如果AMS想要通知ActivityThread做一些事情，应该咋办呢？

还是通过Binder通信，不过是换了另外一对，换成了ApplicationThread和ApplicationThreadProxy。

客户端： ApplicationThread <===== Binder驱动<=====
ApplicationThreadProxy： 服务器

他们也都实现了相同的接口IAplicationThread

```
private class ApplicationThread extends  
ApplicationThreadNative {}  
  
public abstract class ApplicationThreadNative extends  
Binder implements IApplicationThread{}  
  
class ApplicationThreadProxy implements  
IApplicationThread {}
```

剩下的就不必多说了吧，和前面一样。

AMS接收到客户端的请求之后，会如何开启一个Activity？

OK，至此，点击桌面图标调用startActivity()，终于把数据和要开启Activity的请求发送到了AMS了。说了这么多，其实这些都在一瞬间完成了，下面咱们研究下AMS到底做了什么。

注：前方有高能的方法调用链，如果你现在累了，请先喝杯咖啡或者是上趟厕所休息下

AMS收到startActivity的请求之后，会按照如下的方法链进行调用

调用startActivity()

```
@Override  
public final int startActivity(IApplicationThread  
caller, String callingPackage,  
        Intent intent, String resolvedType, IBinder  
resultTo, String resultWho, int requestCode,  
        int startFlags, ProfilerInfo profilerInfo,  
Bundle options) {  
    return startActivityAsUser(caller,  
callingPackage, intent, resolvedType, resultTo,  
        resultWho, requestCode, startFlags,  
profilerInfo, options,  
        UserHandle.getCallingUserId());  
}
```

调用startActivityAsUser()

```
@Override
    public final int
startActivityAsUser(IApplicationThread caller, String
callingPackage,
                    Intent intent, String resolvedType, IBinder
resultTo, String resultWho, int requestCode,
                    int startFlags, ProfilerInfo profilerInfo,
Bundle options, int userId) {

    ...ignore some code...

    return
mStackSupervisor.startActivityMaywait(caller, -1,
callingPackage, intent,
                    resolvedType, null, null, resultTo,
resultWho, requestCode, startFlags,
                    profilerInfo, null, null, options,
userId, null, null);
}
```

在这里又出现了一个新对象ActivityStackSupervisor，通过这个类可以实现对ActivityStack的部分操作。

```
final int startActivityMaywait(IApplicationThread
caller, int callingUid,
                    String callingPackage, Intent intent, String
resolvedType,
                    IVoiceInteractionSession voiceSession,
IVoiceInteractor voiceInteractor,
                    IBinder resultTo, String resultWho, int
requestCode, int startFlags,
                    ProfilerInfo profilerInfo, WaitResult
outResult, Configuration config,
                    Bundle options, int userId,
IActivityContainer iContainer, TaskRecord inTask) {

    ...ignore some code...
```

```
        int res = startActivityLocked(caller,
intent, resolvedType, aInfo,
                voiceSession, voiceInteractor,
resultTo, resultwho,
                requestCode, callingPid, callingUid,
callingPackage,
                realCallingPid, realCallingUid,
startFlags, options,
                componentSpecified, null, container,
inTask);

        ...ignore some code...

    }
```

继续调用startActivityLocked()

```
final int startActivityLocked(IApplicationThread caller,
                                Intent intent, String resolvedType,
ActivityInfo aInfo,
                                IVoiceInteractionSession voicesession,
IVoiceInteractor voiceInteractor,
                                IBinder resultTo, String resultwho, int
requestCode,
                                int callingPid, int callinguid, String
callingPackage,
                                int realCallingPid, int realCallingUid, int
startFlags, Bundle options,
                                boolean componentSpecified, ActivityRecord[]
outActivity, ActivityContainer container,
                                TaskRecord inTask) {

        err = startActivityUncheckedLocked(r,
sourceRecord, voiceSession, voiceInteractor,
                startFlags, true, options, inTask);
        if (err < 0) {
            notifyActivityDrawnForKeyguard();
        }
    }
```

```
    }
    return err;
}
```

调用startActivityUncheckedLocked(),此时要启动的Activity已经通过检验，被认为是一个正当的启动请求。

终于，在这里调用到了ActivityStack的
startActivityLocked(ActivityRecord r, boolean newTask,boolean
doResume, boolean keepCurTransition, Bundle options)。

ActivityRecord代表的就是要开启的Activity对象，里面分装了很多信息，比如所在的ActivityTask等，如果这是首次打开应用，那么这个Activity会被放到ActivityTask的栈顶，

```
final int startActivityUncheckedLocked(ActivityRecord r,
ActivityRecord sourceRecord,
    IVoiceInteractionSession voicesession,
IVoiceInteractor voiceInteractor, int startFlags,
    boolean doResume, Bundle options, TaskRecord
inTask) {

    ...ignore some code...

    targetStack.startActivityLocked(r, newTask,
doResume, keepCurTransition, options);

    ...ignore some code...

    return ActivityManager.START_SUCCESS;
}
```

调用的是ActivityStack.startActivityLocked()

```
final void startActivityLocked(ActivityRecord r, boolean
newTask,
    boolean doResume, boolean keepCurTransition,
Bundle options) {
```

```
//ActivityRecord中存储的TaskRecord信息
TaskRecord rTask = r.task;

...ignore some code...

//如果不是在新的ActivityTask(也就是TaskRecord)中的话,
就找出要运行在的TaskRecord对象
TaskRecord task = null;
if (!newTask) {
    boolean startIt = true;
    for (int taskNdx = mTaskHistory.size() - 1;
taskNdx >= 0; --taskNdx) {
        task = mTaskHistory.get(taskNdx);
        if (task.getTopActivity() == null) {
            // task中的所有Activity都结束了
            continue;
        }
        if (task == r.task) {
            // 找到了
            if (!startIt) {
                task.addActivityToTop(r);
                r.putInHistory();
            }
            mwindowManager.addAppToken(task.mActivities.indexOf(r),
r.appToken,
                                r.task.taskId, mStackId,
r.info.screenOrientation, r.fullscreen,
                                (r.info.flags &
ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0,
                                r.userId,
r.info.configChanges, task.voiceSession != null,
                                r.mLaunchTaskBehind);
            if (VALIDATE_TOKENS) {
                validateAppTokensLocked();
            }
            ActivityOptions.abort(options);
            return;
        }
    }
}
```

```
        break;
    } else if (task.numFullscreen > 0) {
        startIt = false;
    }
}

...
...ignore some code...

// Place a new activity at top of stack, so it is
next to interact
// with the user.
task = r.task;
task.addActivityToTop(r);
task.setFrontOfTask();

...
...ignore some code...

if (doResume) {

mStackSupervisor.resumeTopActivitiesLocked(this, r,
options);
}
}
```

靠！这来回折腾什么呢！从ActivityStackSupervisor到ActivityStack，又调回ActivityStackSupervisor，这到底是在折腾什么玩意啊！！！

淡定...淡定...我知道你也在心里骂娘，世界如此美妙，你却如此暴躁，这样不好，不好...

来来来，咱们继续哈，刚才说到哪里了？哦，对，咱们一起看下
StackSupervisor.resumeTopActivitiesLocked(this, r, options)

```
boolean resumeTopActivitiesLocked(ActivityStack
targetStack, ActivityRecord target,
        Bundle targetOptions) {
    if (targetStack == null) {
        targetStack = getFocusedStack();
```

```
    }

    // Do targetStack first.
    boolean result = false;
    if (isFrontStack(targetStack)) {
        result =
targetStack.resumeTopActivityLocked(target,
targetOptions);
    }

    ...ignore some code...

    return result;
}
```

我...已无力吐槽了，又调回ActivityStack去了...

ActivityStack.resumeTopActivityLocked()

```
final boolean resumeTopActivityLocked(ActivityRecord
prev, Bundle options) {
    if (inResumeTopActivity) {
        // Don't even start recursing.
        return false;
    }

    boolean result = false;
    try {
        // Protect against recursion.
        inResumeTopActivity = true;
        result = resumeTopActivityInnerLocked(prev,
options);
    } finally {
        inResumeTopActivity = false;
    }
    return result;
}
```

咱们坚持住，看一下ActivityStack.resumeTopActivityInnerLocked()到底进行了什么操作

```
final boolean
resumeTopActivityInnerLocked(ActivityRecord prev, Bundle
options) {

    ...ignore some code...
    //找出还没结束的首个ActivityRecord
    ActivityRecord next =
topRunningActivityLocked(null);

    //如果一个没结束的Activity都没有，就开启Launcher程序
    if (next == null) {
        ActivityOptions.abort(options);
        if (DEBUG_STATES) Slog.d(TAG,
"resumeTopActivityLocked: No more activities go home");
        if (DEBUG_STACK)
mStackSupervisor.validateTopActivitiesLocked();
        // Only resume home if on home display
        final int returnTaskType = prevTask == null
|| !prevTask.isOverHomeStack() ?
            HOME_ACTIVITY_TYPE :
prevTask.getTaskToReturnTo();
        return isOnHomeDisplay() &&

mStackSupervisor.resumeHomeStackTask(returnTaskType,
prev);
    }

    //先需要暂停当前的Activity。因为我们是在Launcher中启动
mainActivity，所以当前mResumedActivity!=null，调用
startPausingLocked()使得Launcher进入Pausing状态
    if (mResumedActivity != null) {
        pausing |= startPausingLocked(userLeaving,
false, true, dontwaitForPause);
        if (DEBUG_STATES) Slog.d(TAG,
"resumeTopActivityLocked: Pausing " + mResumedActivity);
    }
}
```

```
    }  
  
}
```

在这个方法里，prev.app为记录启动Lancher进程的ProcessRecord，prev.app.thread为Lancher进程的远程调用接口IAplicationThread，所以可以调用prev.app.thread.schedulePauseActivity，到Lancher进程暂停指定Activity。

```
final boolean startPausingLocked(boolean userLeaving,  
boolean uisleeping, boolean resuming,  
        boolean dontwait) {  
    if (mPausingActivity != null) {  
        completePauseLocked(false);  
    }  
  
    ...ignore some code...  
  
    if (prev.app != null && prev.app.thread != null)  
        try {  
            mService.updateUsageStats(prev, false);  
  
            prev.app.thread.schedulePauseActivity(prev.appToken,  
            prev.finishing,  
                    userLeaving,  
            prev.configChangeFlags, dontwait);  
        } catch (Exception e) {  
            mPausingActivity = null;  
            mLastPausedActivity = null;  
            mLastNoHistoryActivity = null;  
        }  
    } else {  
        mPausingActivity = null;  
        mLastPausedActivity = null;  
        mLastNoHistoryActivity = null;  
    }  
  
    ...ignore some code...
```

```
}
```

在Launcher进程中消息传递，调用ActivityThread.handlePauseActivity()，最终调用ActivityThread.performPauseActivity()暂停指定Activity。接着通过前面所说的Binder通信，通知AMS已经完成暂停的操作。

```
ActivityManagerNative.getDefault().activityPaused(token).
```

上面这些调用过程非常复杂，源码中各种条件判断让人眼花缭乱，所以说如果你没记住也没关系，你只要记住这个流程，理解了Android在控制Activity生命周期时是如何操作，以及是通过哪几个关键的类进行操作的就可以了，以后遇到相关的问题之道从哪块下手即可，这些过程我虽然也是撸了一遍，但还是记不清。

最后来一张高清无码大图，方便大家记忆：

[请戳这里\(图片3.3M，请用电脑观看\)](#)

送给你们的彩蛋

不要使用 startActivityForResult(intent,RESULT_OK)

这是因为startActivity()是这样实现的

```
public void startActivityForResult(Intent intent, @Nullable Bundle options) {
    if (options != null) {
        startActivityForResult(intent, -1, options);
    } else {
        // Note we want to go through this call for
        // compatibility with
        // applications that may have overridden the
        // method.
        startActivityForResult(intent, -1);
    }
}
```

而

```
public static final int RESULT_OK = -1;
```

所以

```
startActivityForResult(intent,RESULT_OK) =  
startActivity()
```

你不可能从onActivityResult()里面收到任何回调。而这个问题是相当难以被发现的，就是因为这个坑，我工作一年多来第一次加班到9点(^-^)

一个App的程序入口到底是什么？

是ActivityThread.main()。

整个App的主线程的消息循环是在哪里创建的？

是在ActivityThread初始化的时候，就已经创建消息循环了，所以在主线程里面创建Handler不需要指定Looper，而如果在其他线程使用Handler，则需要单独使用Looper.prepare()和Looper.loop()创建消息循环。

```
public static void main(String[] args) {  
  
    ...ignore some code...  
  
    Looper.prepareMainLooper();  
  
    ActivityThread thread = new ActivityThread();  
    thread.attach(false);  
  
    if (sMainThreadHandler == null) {  
        sMainThreadHandler = thread.getHandler();  
    }  
  
    AsyncTask.init();  
  
    if (false) {  
        Looper.myLooper().setMessageLogging(new
```

```
        LogPrinter(Log.DEBUG,
    "ActivityThread"));
}

Looper.loop();

    ...ignore some code...

}
```

Application是在什么时候创建的? onCreate()什么时候调用的?

也是在ActivityThread.main()的时候，再具体点呢，就是在
thread.attach(false)的时候。

看你的表情，不信是吧！凯子哥带你溜溜~

我们先看一下ActivityThread.attach()

```
private void attach(boolean system) {
    mCurrentActivityThread = this;
    mSystemThread = system;
    //普通App进这里
    if (!system) {

        ...ignore some code...
```

```
RuntimeInit.setApplicationObject(mAppThread.asBinder());
    final IActivityManager mgr =
ActivityManagerNative.getDefault();
    try {
        mgr.attachApplication(mAppThread);
    } catch (RemoteException ex) {
        // Ignore
    }
} else {
    //这个分支在SystemServer加载的时候会进入，通过调用
    // private void createSystemContext() {
```

```
//      ActivityThread activityThread =
ActivityThread.systemMain();
//}

// public static ActivityThread systemMain()
{
    //      if (!ActivityManager.isHighEndGfx()) {
    //          HardwareRenderer.disable(true);
    //      } else {
    //
HardwareRenderer.enableForegroundTrimming();
    //      }
    //      ActivityThread thread = new
ActivityThread();
    //      thread.attach(true);
    //      return thread;
    //  }
}

}
```

这里需要关注的就是mgr.attachApplication(mAppThread)，这个就会通过Binder调用到AMS里面对应的方法

```
@Override
public final void
attachApplication(IApplicationThread thread) {
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final long origId =
Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid);
        Binder.restoreCallingIdentity(origId);
    }
}
```

然后就是

```
private final boolean  
attachApplicationLocked(IApplicationThread thread,  
        int pid) {  
  
        thread.bindApplication(processName, appInfo,  
providers, app.instrumentationClass,  
                profilerInfo,  
app.instrumentationArguments, app.instrumentationWatcher,  
  
app.instrumentationUiAutomationConnection, testMode,  
enableOpenGLTrace,  
                isRestrictedBackupMode ||  
!normalMode, app.persistent,  
                new Configuration(mConfiguration),  
app.compat, getCommonServicesLocked(),  
  
mCoreSettingsObserver.getCoreSettingsLocked());  
  
    }  
}
```

thread是IApplicationThread，实际上就是ApplicationThread在服务端的代理类ApplicationThreadProxy，然后又通过IPC就会调用到ApplicationThread的对应方法

```
private class ApplicationThread extends  
ApplicationThreadNative {  
  
    public final void bindApplication(String processName,  
ApplicationInfo appInfo,  
            List<ProviderInfo> providers,  
ComponentName instrumentationName,  
            ProfilerInfo profilerInfo, Bundle  
instrumentationArgs,  
            IInstrumentationWatcher  
instrumentationWatcher,  
            IUiAutomationConnection  
instrumentationUiConnection, int debugMode,
```

```
        boolean enableOpenGLTrace, boolean
isRestrictedBackupMode, boolean persistent,
        Configuration config, CompatibilityInfo
compatInfo, Map<String, IBinder> services,
        Bundle coreSettings) {

    ...ignore some code...

    AppBindData data = new AppBindData();
    data.processName = processName;
    data.appInfo = appInfo;
    data.providers = providers;
    data.instrumentationName =
instrumentationName;
    data.instrumentationArgs =
instrumentationArgs;
    data.instrumentationWatcher =
instrumentationWatcher;
    data.instrumentationUiAutomationConnection =
instrumentationUiConnection;
    data.debugMode = debugMode;
    data.enableOpenGLTrace = enableOpenGLTrace;
    data.restrictedBackupMode =
isRestrictedBackupMode;
    data.persistent = persistent;
    data.config = config;
    data.compatInfo = compatInfo;
    data.initProfilerInfo = profilerInfo;
    sendMessage(H.BIND_APPLICATION, data);

}

}
```

我们需要关注的其实就是最后的sendMessage(), 里面有函数的编号H.BIND_APPLICATION, 然后这个Message会被H这个Handler处理

```
private class H extends Handler {
```

```
    ...ignore some code...

    public static final int BIND_APPLICATION      =
110;

    ...ignore some code...

    public void handleMessage(Message msg) {
        switch (msg.what) {
        ...ignore some code...
        case BIND_APPLICATION:

Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"bindApplication");
        AppBindData data =
(AppBindData)msg.obj;
        handleBindApplication(data);

Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        break;
        ...ignore some code...
    }
}
```

最后就在下面这个方法中，完成了实例化，拔那个企鹅通过
mInstrumentation.callApplicationOnCreate实现了onCreate()的调用。

```
private void handleBindApplication(AppBindData data) {

try {

    ...ignore some code...

    Application app =
data.info.makeApplication(data.restrictedBackupMode,
null);
    mInitialApplication = app;
```

```
    ...ignore some code...

    try {
        mInstrumentation.onCreate(data.instrumentationArgs);
    }
    catch (Exception e) {
    }
    try {
        mInstrumentation.callApplicationOnCreate(app);
    } catch (Exception e) {
    }
    } finally {
        StrictMode.setThreadPolicy(savedPolicy);
    }
}
```

data.info是一个LoadApk对象。

LoadApk.data.info.makeApplication()

```
public Application makeApplication(boolean
forceDefaultAppClass,
        Instrumentation instrumentation) {
    if (mApplication != null) {
        return mApplication;
    }

    Application app = null;

    String appClass = mApplicationInfo.className;
    if (forceDefaultAppClass || (appClass == null)) {
        appClass = "android.app.Application";
    }

    try {
        java.lang.ClassLoader cl = getClassLoader();
        if (!mPackageName.equals("android")) {

```

```
        initializeJavaContextClassLoader();
    }
    ContextImpl appContext =
ContextImpl.createApplicationContext(mActivityThread, this);
    app =
mActivityThread.mInstrumentation.newApplication(
        c1, appClass, appContext);
    appContext.setOuterContext(app);
} catch (Exception e) {
}
mActivityThread.mAllApplications.add(app);
mApplication = app;

//传进来的是null，所以这里不会执行，onCreate在上一层执行
if (instrumentation != null) {
    try {
instrumentation.callApplicationOnCreate(app);
    } catch (Exception e) {
    }
}
...ignore some code...

}
return app;
}
```

所以最后还是通过Instrumentation.makeApplication()实例化的，这个老板娘真的很厉害呀！

```
static public Application newApplication(Class<?> clazz,  
Context context)  
        throws InstantiationException,  
IllegalAccessException,  
ClassNotFoundException {  
    Application app =  
(Application)clazz.newInstance();  
    app.attach(context);  
    return app;  
}
```

而且通过反射拿到Application对象之后，直接调用attach()，所以attach()调用是在onCreate()之前的。

1.2启动模式以及使用场景

1.3onSaveInstanceState以及 onRestoreInstanceState使用

1.4onConfigurationChanged使用以及问题解决

第二节 Fragment知识点

- Fragment生命周期
- Fragment的懒加载
- Fragment之间的通信
- FragmentPagerAdapter 和FragmentStatePagerAdapter区别

2.1 Fragment的通信问题, 新建Fragment为何不要在构造方法中传递参数

最近有个新进来的同事做Android开发,因为之前都是做Java的,所以新建对象习惯在构造方法里面去传递参数回调方法什么的。

于是当他在Activity中创建Fragment的时候,也使用了类似new TestFragment(“content”)这样的方式去将Activity中的参数传递到TestFragment中。

第一次用的时候没报错,然后进出几次TestFragment就报 android.support.v4.app.Fragment\$InstantiationException 错了. 然后满脸疑惑的问怎么了?

如果将Fragment换做是普通的对象,比如是自己自定义的bean对象,在构造方法中传递参数是没问题的.

但是Fragment和Activity一样,是有生命周期的组件,并不能看做是一般的类.于是我说你在构造方法中不要直接用类似的方式去写:

```
public static TestFragment getInstance(String tokenId) {  
    if (null == sInstance) {  
        synchronized (TestFragment.class) {  
            if (sInstance == null) {  
                sInstance = new TestFragment();  
            }  
        }  
    }  
    tokenId = tokenId;  
    return sInstance;  
}
```

第一次传递tokenId这个参数可能在Fragment中是OK的.但到后面tokenId就可能为空了. 为什么呢?

- Fragment的生命周期依附在Activity中,如果Activity为null,那么Fragment肯定要出事儿.
- 手机屏幕竖屏横屏切换,导致Activity重建了,于是Fragment中的所有原先传递过去的值也会失去.也就是说tokenId这个时候是空的,或者变为

原本的默认值.

这里就涉及了Activity和Fragment之间如何通信的情况.

当然可以用接口什么的,但这里使用fragment.setArguments(bundle)去实现.

Bundle是一个很好的用于传递参数的工具对象.并且在Fragment通过Bundle传递的参数,即使Fragment重建,这个Bundle中的参数也能被保存下来,在新的Fragment中继续用.

至于为什么是这样的呢?我也好奇,于是去看了下Fragment初始化的源码,有这么一段:

```
public static Fragment instantiate(Context context,
String fname, Bundle args) {
    try {
        Class<?> clazz = sClassMap .get(fname);
        if (clazz == null) { // Class not found in the
cache, see if it's real, and try to add it
            clazz =
context.getClassLoader().loadClass(fname);
            sClassMap .put(fname, clazz);
        }
        /*获取Bundle原先的值,这样一开始利用Bundle传递进来的值,就放入
f. mArguments.*/
       只需要在Fragment中利用getArguments().getString("key");就能将参数取出来继续用 */
        Fragment f = (Fragment)clazz.newInstance();
        if (args != null) {

args.setClassLoader(f.getClass().getClassLoader());
        f. mArguments = args;
    }
    return f;
} catch (ClassNotFoundException e) {
    throw new InstantiationException( "Unable to
instantiate fragment " + fname + ": make sure class name
exists, is public, and has an" + " empty constructor that
is public" , e);
}
```

```
    } catch (java.lang.InstantiationException e) {
        throw new InstantiationException("Unable to
        instantiate fragment " + fname + ": make sure class name
        exists, is public, and has an" + " empty constructor that
        is public" , e);
    } catch (IllegalAccessException e) {
        throw new InstantiationException("Unable to
        instantiate fragment " + fname + ": make sure class name
        exists, is public, and has an" + " empty constructor that
        is public" , e);
    } //...
```

整个过程中,Fragment的创建其实也是利用了无参数的构造方法去实例化.但关键的是,它将Bundle传类新建的Fragment,这样旧的Fragment和新的Fragment就能拥有一样的Bundle,从而达到利用Bundle传递参数的目的.

既然知道了原理,具体怎么用?当然是利用Bundle和这个setArguments(bundle)方法,在构造Fragment的方法中加入:

```
Bundle bundle = new Bundle(); bundle.putString("key",
value); fragment.setArguments(bundle);
```

在Fragment的周期方法,比如onCreateView()中去取出Bundle就行喇.

```
String value = getArguments().getString("key");
```

这样就大功告成了呢.

2.2 为什么官方推荐Fragment.setArguments(Bundle bundle)这种方式来传递参数,而不推荐通过构造方法直接来传递参数呢?

为什么官方推荐Fragment.setArguments(Bundle bundle)这种方式来传递参数，而不推荐通过构造方法直接来传递参数呢？

看上去这两种方式没有什么本质的区别，但是通过构造方法传递参数的方式是有隐患的。根据Android文档说明，当一个Fragment重新创建的时候，**系统会再次调用Fragment中的默认构造函数，注意是默认构造函数**。即，当你创建了一个带有参数的Fragment的之后，一旦由于什么原因（例如横竖屏切换）导致你的Fragment重新创建。那么，很遗憾，你之前传递的参数都不见了，因为reCreate你的Fragment的时候，调用的是**默认构造函数**。因此，官方推荐使用Fragment.setArguments(Bundle bundle)这种方式来传递参数，而不推荐通过构造方法直接来传递参数。

2.3 Androidx 下 Fragment 懒加载的新实现

前言

年后最后一篇文章啦，在这里先祝大家新年快乐~最重要的抽中[全家福](#)，明年继续修福报

以前处理 Fragment 的懒加载，我们通常会在 Fragment 中处理 `setUserVisibleHint + onHiddenChanged` 这两个函数，而在 Androidx 模式下，我们可以使用 `FragmentTransaction.setMaxLifecycle()` 的方式来处理 Fragment 的懒加载。

在本文章中，我会详细介绍不同使用场景下两种方案的差异。大家快拿好小板凳。一起来学习新知识吧！

本篇文章涉及到的 Demo，已上传至Github—>[传送门](#)

老的懒加载处理方案

如果你熟悉老一套的 Fragment 懒加载机制，你可以直接查看 Androidx 懒加载相关章节

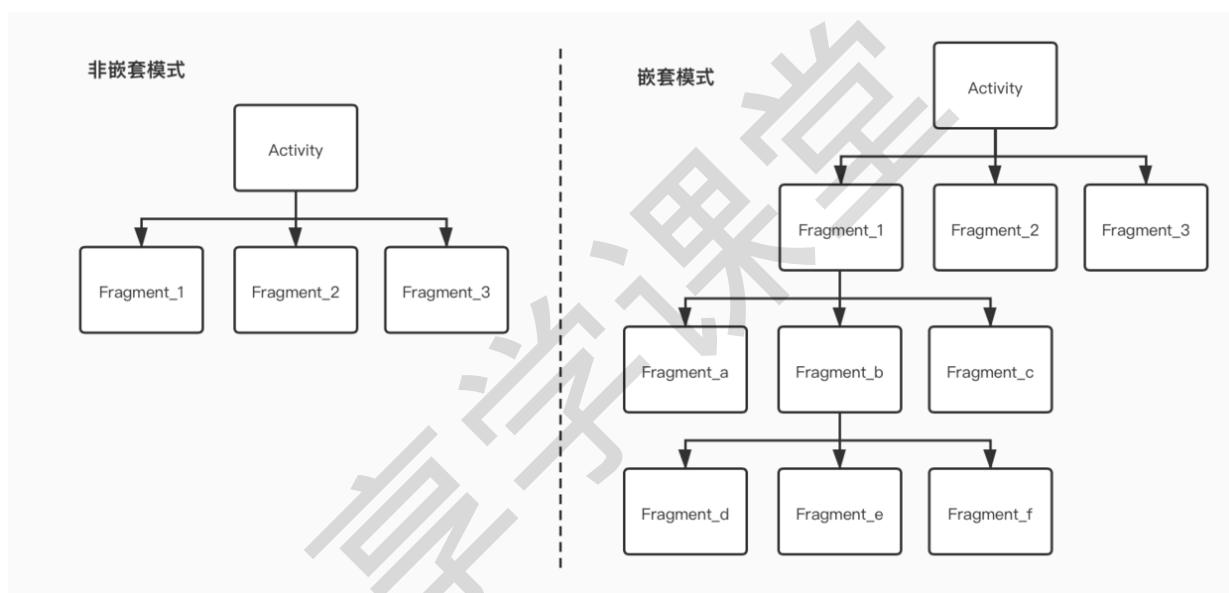
add+show+hide 模式下的老方案

在没有添加懒加载之前，只要使用 `add+show+hide` 的方式控制并显示 Fragment，那么不管 Fragment 是否嵌套，在初始化后，如果只调用了 `add+show`，同级下的 Fragment 的相关生命周期函数都会被调用。且调用的生命周期函数如下所示：

```
onAttach -> onCreate -> onCreateView -> onActivityCreated ->  
onStart -> onResume
```

Fragment 完整生命周期： `onAttach -> onCreate -> onCreateView -> onActivityCreated -> onStart -> onResume -> onPause -> onStop -> onDestroyView -> onDestroy -> onDetach`

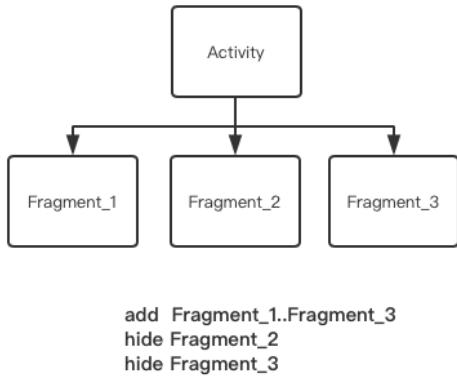
什么是同级 Fragment 呢？看下图



上图中，都是使用 `add+show+hide` 的方式控制 Fragment，在上图两种模式中：

- Fragment_1、Fragment_2、Fragment_3 属于同级 Fragment
- Fragment_a、Fragment_b、Fragment_c 属于同级 Fragment
- Fragment_d、Fragment_e、Fragment_f 属于同级 Fragment

那这种方式会带来什么问题呢？结合下图我们来分别分析。



```

OneFragment: onAttach:  

OneFragment: onCreate:  

TwoFragment: onAttach:  

TwoFragment: onCreate:  

ThreeFragment: onAttach:  

ThreeFragment: onCreate:  

TwoFragment: onViewCreated:  

TwoFragment: onActivityCreated:  

TwoFragment: onHiddenChanged:hidden-->true  

ThreeFragment: onViewCreated:  

ThreeFragment: onActivityCreated:  

ThreeFragment: onHiddenChanged:hidden-->true  

OneFragment: onViewCreated:  

OneFragment: onActivityCreated:  

OneFragment: onStart:  

TwoFragment: onStart:  

ThreeFragment: onStart:  

OneFragment: onResume:  

TwoFragment: onResume:  

ThreeFragment: onResume:

```

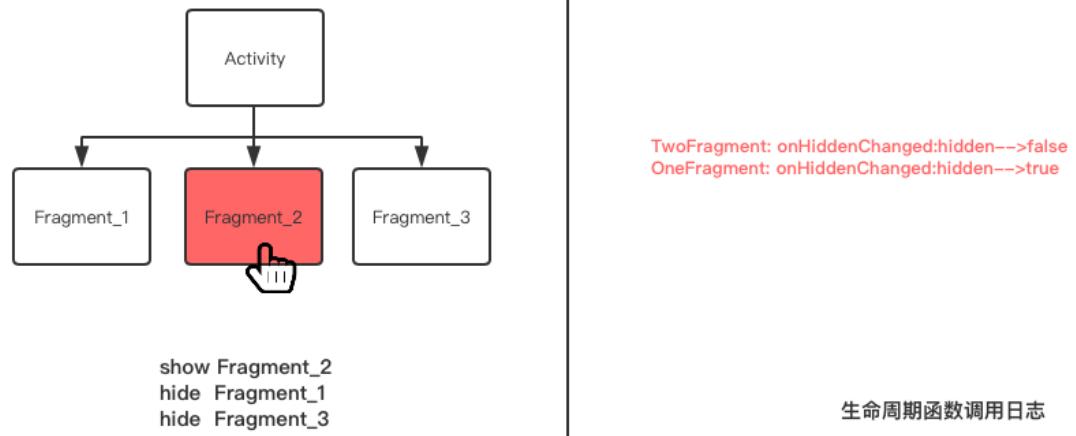
生命周期函数调用日志

观察上图我们可以发现，同级的 Fragment_1、Fragment_2、Fragment_3 都调用了 `onAttach...onResume` 系列方法，也就是说，如果我们没有对 Fragment 进行懒加载处理，那么我们就会无缘无故的加载一些并不可见的 Fragment，也就会造成用户流量的无故消耗（我们会在 Fragment 相关生命周期函数中，请求网络或其他数据操作）。

这里 "不可见的Fragment" 是指，实际不可见但是相关可见生命周期函数(如 `onResume` 方法) 被调用的 Fragment

如果使用嵌套 Fragment，这种浪费流量的行为就更明显了。以本节的图一为例，当 Fragment_1 加载时，如果你在 Fragment_1 生命周期函数中使用 `show+add+hide` 的方式添加 Fragment_a、Fragment_b、Fragment_c，那么 Fragment_b 又会在其生命周期函数中继续加载 Fragment_d、Fragment_e、Fragment_f。

那如何解决这种问题呢？我们继续接着上面的例子走，当我们 `show Fragment_2`，并 `hide` 其他 Fragment 时，对应 Fragment 的生命周期调用如下：



从上图中，我们可以看出 Fragment_2 与 Fragment_3 都调用了 `onHiddenChanged` 函数，该函数的官方 API 声明如下：

| |
|--|
| <pre> 1 /** 2 * called when the hidden state (as returned by 3 * {@link #isHidden()}) of 4 * the fragment has changed. Fragments start 5 * out not hidden; this will 6 * be called whenever the fragment changes state 7 * from that. 8 * 9 * @param hidden True if the fragment is now 10 * hidden, false otherwise. 11 */ 12 public void onHiddenChanged(boolean hidden) { 13 } </pre> |
|--|

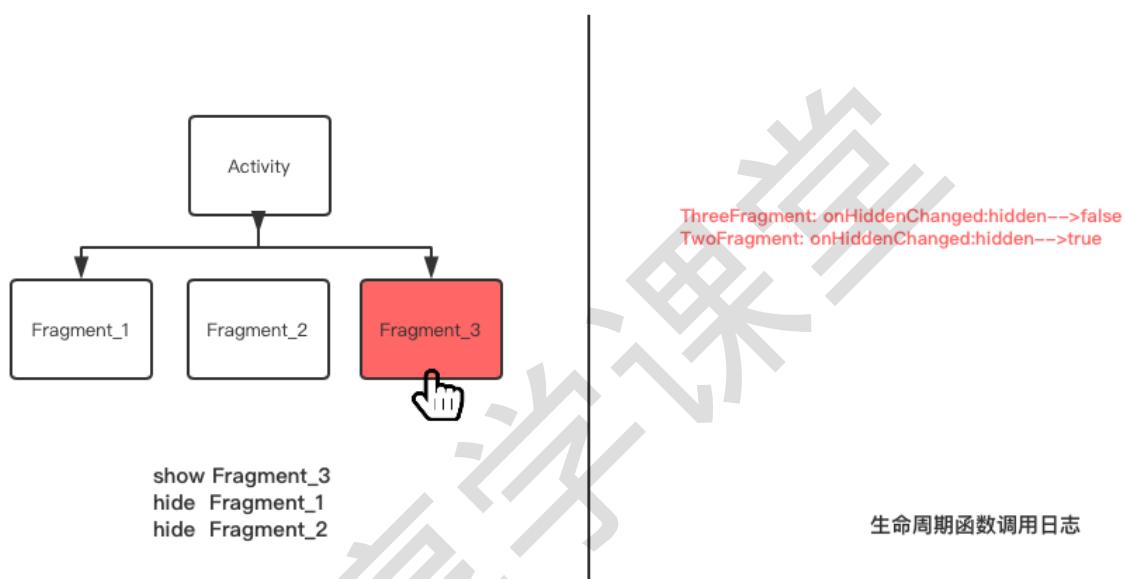
根据官方 API 的注释，我们大概能知道，当 Fragment 隐藏的状态发生改变时，该函数将会被调用，如果当前 Fragment 隐藏，`hidden` 的值为 `true`，反之为 `false`。最为重要的是 `hidden` 的值，可以通过调用 `isHidden()` 函数获取。

那么结合上述知识点，我们能推导出：

- 因为 Fragment_1 的 `隐藏状态` 从可见转为了不可见，所以其 `onHiddenChanged` 函数被调用，同时 `hidden` 的值为 true。
- 同理对于 Fragment_2，因为其 `隐藏状态` 从不可见转为了可见，所以其 `hidden` 值为 false。
- 对于 Fragment_3，因为其 `隐藏状态` 从始至终都没有发生变化，所以其 `onHiddenChanged` 函数并不会调用。

嗯，好像有点眉目了。不急，我们继续看下面的例子。

show Fragment_3 并 hide 其他 Fragment，对应生命周期函数调用如下所示：



从图中，我们可以看出，确实只有 `隐藏状态` 发生了改变的 Fragment 其 `onHiddenChanged` 函数才会调用，那么结合以上知识点，我们能得出如下重要结论：

只要通过 `show+hide` 方式控制 Fragment 的显隐，那么在第一次初始化后，Fragment 任何的生命周期方法都不会调用，只有 `onHiddenChanged` 方法会被调用。

那么，假如我们要在 `add+show+hide` 模式下控制 Fragment 的懒加载，我们只需要做这两步：

- 我们需要在 `onResume()` 函数中调用 `isHidden()` 函数，来处理默认显示的 Fragment
- 在 `onHiddenChanged` 函数中控制其他不可见的 Fragment，

也就是这样处理：

題解詳解

```
1 abstract class LazyFragment:Fragment(){
2     private var isLoaded = false //控制是否执行懒
3     加载
4
5     override fun onResume() {
6         super.onResume()
7         judgeLazyInit()
8
9     }
10    override fun onHiddenChanged(hidden:
11        Boolean) {
12        super.onHiddenChanged(hidden)
13        isVisibleToUser = !hidden
14        judgeLazyInit()
15    }
16
17    private fun judgeLazyInit() {
18        if (!isLoaded && !isHidden) {
19            lazyInit()
20            isLoaded = true
21        }
22    }
23
24    override fun onDestroyView() {
25        super.onDestroyView()
26        isLoaded = false
27    }
28
29    //懒加载方法
30    abstract fun lazyInit()
}
```

该懒加载的实现，是在 `onResume` 方法中操作，当然你可以在其他生命周期函数中控制。但是建议在该方法中执行懒加载。

ViewPager+Fragment 模式下的老方案

使用传统方式处理 ViewPager 中 Fragment 的懒加载，我们需要控制 `setUserVisibleHint(boolean isVisibleToUser)` 函数，该函数的声明如下所示：

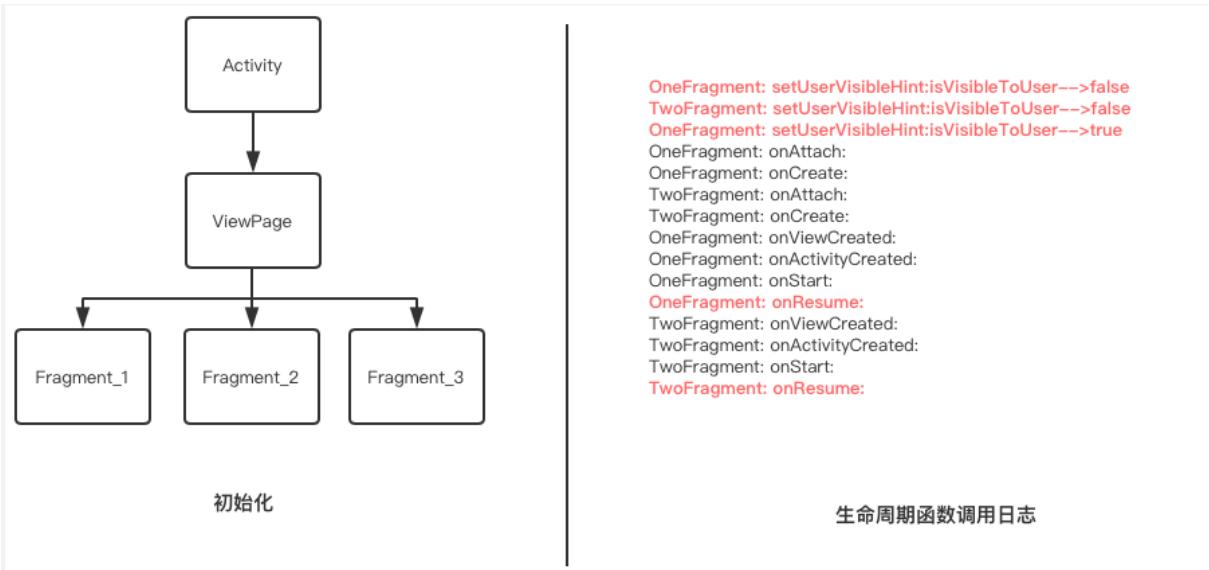
```
1 public void setUserVisibleHint(boolean  
    isVisibleToUser) {}
```

该函数与之前我们介绍的 `onHiddenChanged()` 作用非常相似，都是通过传入的参数值来判断当前 Fragment 是否对用户可见，只是 `onHiddenChanged()` 是在 `add+show+hide` 模式下使用，而 `setUserVisibleHint` 是在 ViewPager+Fragment 模式下使用。

在本节中，我们用 `FragmentPagerAdapter + viewPager` 为例，向大家讲解如何实现 Fragment 的懒加载。

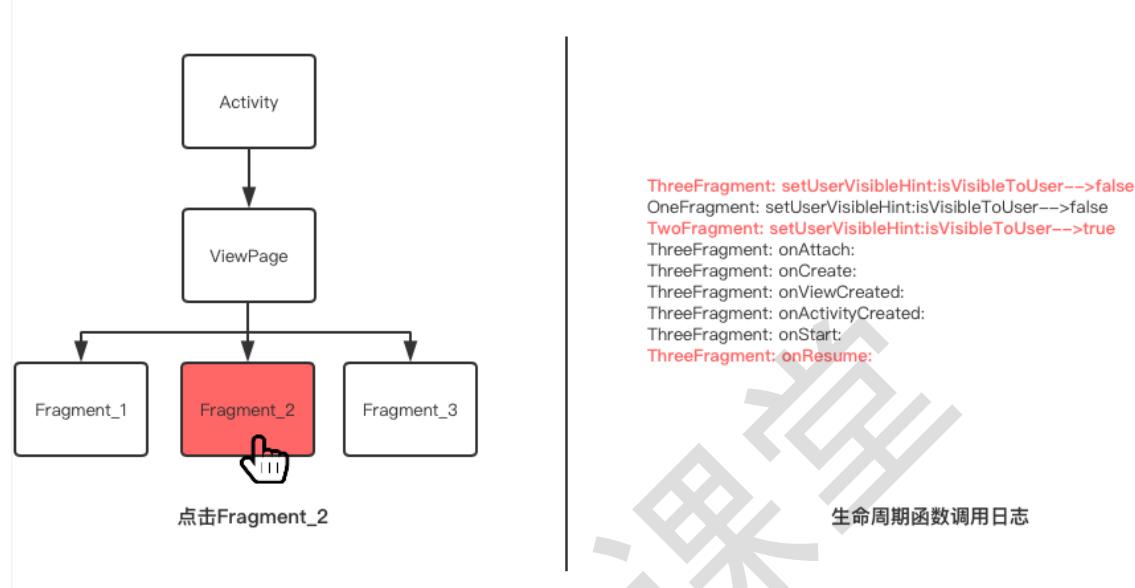
注意：在本例中没有调用 `setOffscreenPageLimit` 方法去设置 ViewPager 预缓存的 Fragment 个数。默认情况下 ViewPager 预缓存 Fragment 的个数为 1。

初始化 ViewPager 查看内部 Fragment 生命周期函数调用情况：

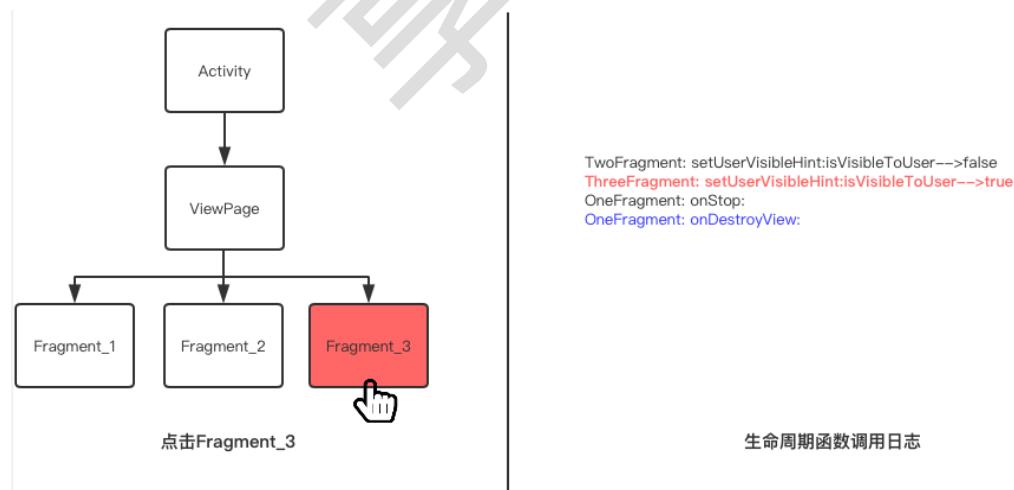


观察上图，我们能发现 ViePager 初始化时，默认会调用其内部 Fragment 的 setUserVisibleHint 方法，因为其预缓存 Fragment 个数为 1 的原因，所以只有 Fragment_1 与 Fragment_2 的生命周期函数被调用。

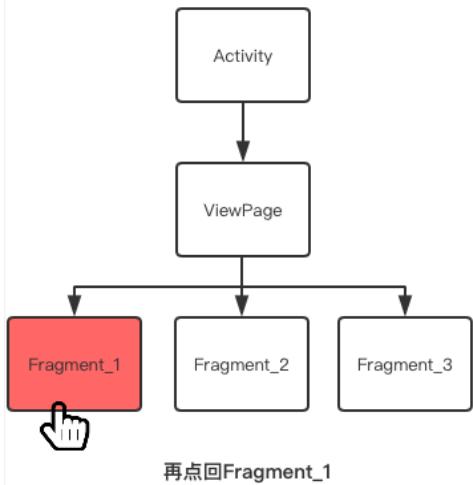
我们继续切换到 Fragment_2，查看各个Fragment的生命周期函数的调用变化。如下图所示：



观察上图，我们同样发现 Fragment 的 setUserVisibleHint 方法被调用了，并且 Fragment_3 的一系列生命周期函数被调用了。继续切换到 Fragment_3：



观察上图可以发现，Fragment_3 调用了 setUserVisibleHint 方法，继续又切换到 Fragment_1，查看调用函数的变化：



```

OneFragment: setUserVisibleHint:isVisibleToUser-->false
ThreeFragment: setUserVisibleHint:isVisibleToUser-->false
OneFragment: setUserVisibleHint:isVisibleToUser-->true
OneFragment: onViewCreated:
OneFragment: onActivityCreated:
OneFragment: onStart:
OneFragment: onResume:
ThreeFragment: onStop:
ThreeFragment: onDestroyView:

```

生命周期函数调用日志

因为之前在切换到 Fragment_3 时， Fragment_1 已经走了
onDestoryView(图二，蓝色标记处) 方法，所以 Fragment_1 需要重
新走一次生命周期。

那么结合本节的三幅图，我们能得出以下结论：

- 使用 ViewPager，切换回上一个 Fragment 页面时（已经初始化完
毕），不会回调任何生命周期方法以及 onHiddenChanged()，只有
setUserVisibleHint(boolean isVisibleToUser) 会被回调。
- setUserVisibleHint(boolean isVisibleToUser) 方法总是会优先于
Fragment 生命周期函数的调用。

所以如果我们想对 ViewPager 中的 Fragment 懒加载，我们需要这样处
理：

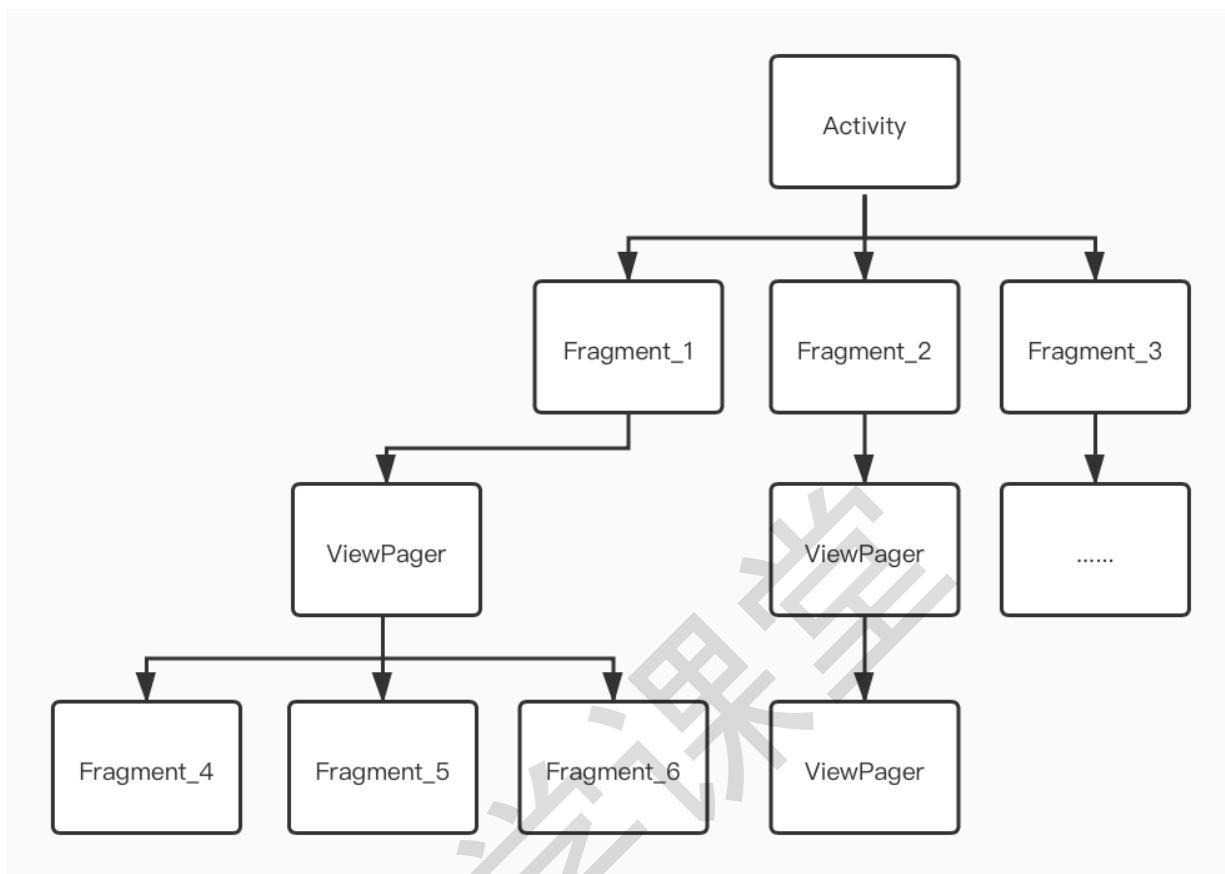
```
1 abstract class LazyFragment : Fragment() {
2
3     /**
4      * 是否执行懒加载
5      */
6     private var isLoaded = false
7
8     /**
9      * 当前Fragment是否对用户可见
10     */
11    private var isVisibleToUser = false
12
13    /**
14     * 当使用ViewPager+Fragment形式会调用该方法时,
15     setUserVisibleHint会优先Fragment生命周期函数调用,
16     * 所以这个时候就,会导致在setUserVisibleHint方
17     法执行时就执行了懒加载,
18     * 而不是在onResume方法实际调用的时候执行懒加载。
19     所以需要这个变量
20     */
21    private var isCallResume = false
22
23    override fun onResume() {
24        super.onResume()
25        isCallResume = true
26        judgeLazyInit()
27    }
28
29
30    private fun judgeLazyInit() {
31        if (!isLoaded && isVisibleToUser &&
32        isCallResume) {
33            lazyInit()
34            Log.d(TAG, "lazyInit!!!!!!")
```

```
35             isLoading = true
36         }
37     }
38
39     override fun onHiddenChanged(hidden:
40 Boolean) {
41         super.onHiddenChanged(hidden)
42         isVisibleToUser = !hidden
43         judgeLazyInit()
44     }
45
46     //在Fragment销毁view的时候，重置状态
47     override fun onDestroyView() {
48         super.onDestroyView()
49         isLoading = false
50         isVisibleToUser = false
51         isCallResume = false
52     }
53
54     override fun
55     setUserVisibleHint(isVisibleToUser: Boolean) {
56
        super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        judgeLazyInit()
    }
58
59     abstract fun lazyInit()
60 }
```

复杂 Fragment 嵌套的情况

当然，在实际项目中，我们可能会遇到更为复杂的 Fragment 嵌套组合。
比如 Fragment+Fragment、Fragment+ViewPager、
ViewPager+ViewPager....等等。

如下图所示：



对于以上场景，我们就需要重写我们的懒加载，以支持不同嵌套组合模式下 Fragment 正确懒加载。我们需要将 LazyFragment 修改成如下这样：

```
1 abstract class LazyFragment : Fragment() {
2
3     /**
4      * 是否执行懒加载
5      */
6     private var isLoaded = false
7
8     /**
9      * 当前Fragment是否对用户可见
10     */
11    private var isVisibleToUser = false
12
13    /**
14     * 当使用ViewPager+Fragment形式会调用该方法时,
15     setUserVisibleHint会优先Fragment生命周期函数调用,
16     * 所以这个时候就,会导致在setUserVisibleHint方
17     法执行时就执行了懒加载,
18     * 而不是在onResume方法实际调用的时候执行懒加载。
19     所以需要这个变量
20     */
21    private var isCallResume = false
22
23    /**
24     * 是否调用了setUserVisibleHint方法。处理
25     show+add+hide模式下, 默认可见 Fragment 不调用
26     * onHiddenChanged 方法, 进而不执行懒加载方法的
27     问题。
28     */
29    private var isCallUserVisibleHint = false
30
31    override fun onResume() {
32        super.onResume()
33        isCallResume = true
34        if (!isCallUserVisibleHint)
```

```
35    isVisibleToUser = !isHidden
36        judgeLazyInit()
37    }
38
39
40    private fun judgeLazyInit() {
41        if (!isLoading && isVisibleToUser &&
42 isCallResume) {
43            lazyInit()
44            Log.d(TAG, "lazyInit:!!!!!!")
45            isLoading = true
46        }
47    }
48
49    override fun onHiddenChanged(hidden:
50 Boolean) {
51        super.onHiddenChanged(hidden)
52        isVisibleToUser = !hidden
53        judgeLazyInit()
54    }
55
56    override fun onDestroyView() {
57        super.onDestroyView()
58        isLoading = false
59        isVisibleToUser = false
60        isCallUserVisibleHint = false
61        isCallResume = false
62    }
63
64    override fun
setUserVisibleHint(isVisibleToUser: Boolean) {
super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        isCallUserVisibleHint = true
}
```

```
        judgeLazyInit()  
    }  
  
    abstract fun lazyInit()  
}
```

Androidx 下的懒加载

虽然之前的方案就能解决轻松的解决 Fragment 的懒加载，但这套方案有一个最大的弊端，就是不可见的 Fragment 执行了 `onResume()` 方法。`onResume` 方法设计的初衷，难道不是当前 Fragment 可以和用户进行交互吗？你他妈既不可见，又不能和用户进行交互，你执行 `onResume` 方法干嘛？

基于此问题，Google 在 Androidx 在 `FragmentTransaction` 中增加了 `setMaxLifecycle` 方法来控制 Fragment 所能调用的最大的生命周期函数。如下所示：

```
 /**
 * Set a ceiling for the state of an active
fragment in this FragmentManager. If fragment
is
    * already above the received state, it will
be forced down to the correct state.
 *
1   * <p>The fragment provided must currently be
2 added to the FragmentManager to have it's
3     * Lifecycle state capped, or previously added
4 as part of this transaction. The
5     * {@link Lifecycle.State} passed in must at
6 least be {@link Lifecycle.State#CREATED},
7 otherwise
8     * an {@link IllegalArgumentException} will be
9 thrown.</p>
10   *
11   * @param fragment the fragment to have it's
12 state capped.
13   * @param state the ceiling state for the
14 fragment.
15   * @return the same FragmentTransaction
16 instance
17   */
18 @NonNull
19 public FragmentTransaction
setMaxLifecycle(@NonNull Fragment fragment,
    @NonNull Lifecycle.State state) {
    addOp(new Op(OP_SET_MAX_LIFECYCLE,
fragment, state));
    return this;
}
```

根据官方的注释，我们能知道，该方法可以设置活跃状态下 Fragment 最大的状态，如果该 Fragment 超过了设置的最大状态，那么会强制将 Fragment 降级到正确的状态。

那如何使用该方法呢？我们先看该方法在 Androidx 模式下 ViewPager+Fragment 模式下的使用例子。

ViewPager+Fragment 模式下的方案

在 FragmentPagerAdapter 与 FragmentStatePagerAdapter 新增了含有 `behavior` 字段的构造函数，如下所示：

```
1 public FragmentPagerAdapter(@NotNull
2     FragmentManager fm,
3     @Behavior int behavior) {
4     mFragmentManager = fm;
5     mBehavior = behavior;
6 }
7 public FragmentStatePagerAdapter(@NotNull
8     FragmentManager fm,
9     @Behavior int behavior) {
10    mFragmentManager = fm;
11    mBehavior = behavior;
12 }
```

其中 Behavior 的声明如下：

```
1  @Retention(RetentionPolicy.SOURCE)
2      @IntDef({BEHAVIOR_SET_USER_VISIBLE_HINT,
3 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT})
4  private @interface Behavior { }
5
6  /**
7      * Indicates that {@link
8 Fragment#setUserVisibleHint(boolean)} will be
9 called when the current
10     * fragment changes.
11     *
12     * @deprecated This behavior relies on the
13 deprecated
14     * {@link
15 Fragment#setUserVisibleHint(boolean)} API. Use
16     * {@link
17 #BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT} to
18 switch to its replacement,
19     * {@link
20 FragmentTransaction#setMaxLifecycle}.
21     * @see #FragmentPagerAdapter(FragmentManager,
22 int)
23     */
24 @Deprecated
25 public static final int
BEHAVIOR_SET_USER_VISIBLE_HINT = 0;
26
27 /**
28      * Indicates that only the current fragment
29 will be in the {@link Lifecycle.State#RESUMED}
30     * state. All other Fragments are capped at
31 {@link Lifecycle.State#STARTED}.
32     *
33     * @see #FragmentPagerAdapter(FragmentManager,
```

```
    int)
    */
public static final int
BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT = 1;
```

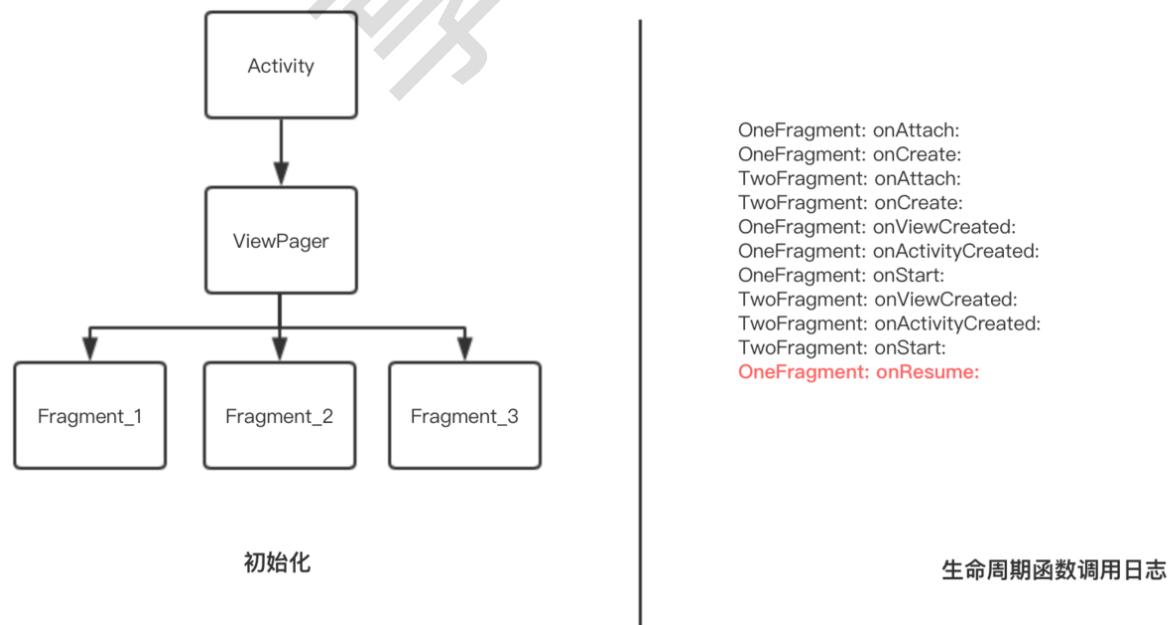
从官方的注释声明中，我们能得到如下两条结论：

- 如果 behavior 的值为 `BEHAVIOR_SET_USER_VISIBLE_HINT`，那么当 Fragment 对用户的可见状态发生改变时，`setUserVisibleHint` 方法会被调用。
- 如果 behavior 的值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`，那么当前选中的 Fragment 在 `Lifecycle.State#RESUMED` 状态，其他不可见的 Fragment 会被限制在 `Lifecycle.State#STARTED` 状态。

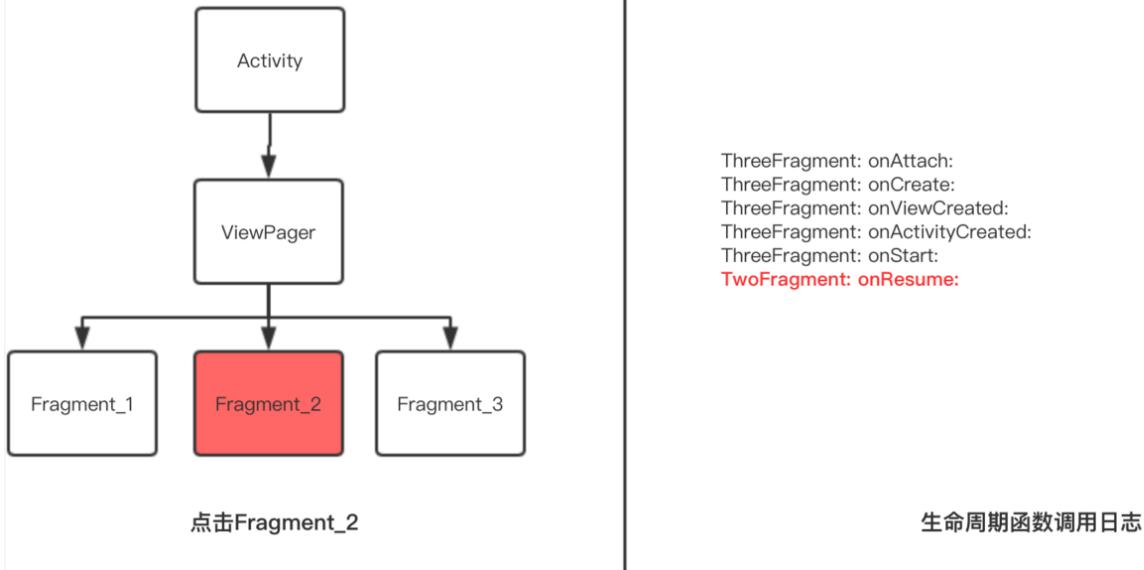
那 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT` 这个值到底有什么作用呢？我们看下面的例子：

在该例子中设置了 ViewPager 的适配器为 FragmentPagerAdapter 且 behavior 值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`。

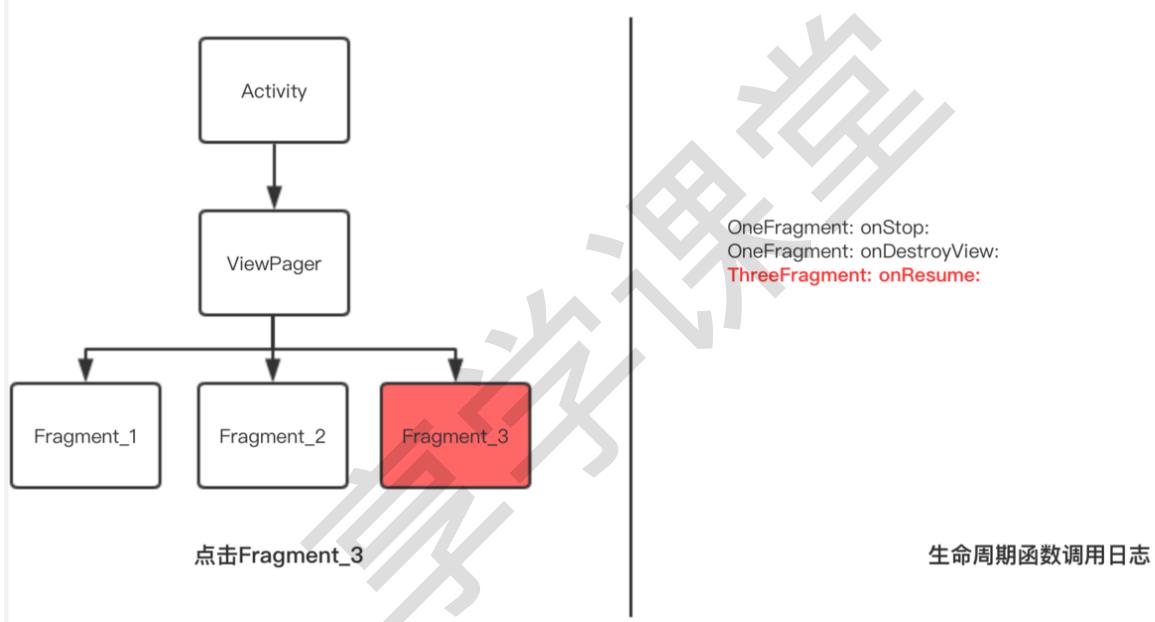
默认初始化ViewPager，Fragment 生命周期如下所示：



切换到 `Fragment_2` 时，日志情况如下所示：



切换到 Fragment_3 时，日志情况如下所示：



因为篇幅的原因，本文没有在讲解 FragmentStatePagerAdapter 设置 behavior 下的使用情况，但是原理以及生命周期函数调用情况一样，感兴趣的小伙伴，可以根据 [AndroidxLazyLoad](#) 项目自行测试。

观察上述例子，我们可以发现，使用了

`BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT` 后，确实只有当前可见的 Fragment 调用了 onResume 方法。而导致产生这种改变的原因，是因为 FragmentPagerAdapter 在其 `setPrimaryItem` 方法中调用了 `setMaxLifecycle` 方法，如下所示：

```
1 public void setPrimaryItem(@NonNull ViewGroup
2 container, int position, @NonNull Object object) {
3     Fragment fragment = (Fragment) object;
4     //如果当前的fragment不是当前选中并可见的Fragment,那么
5     就会调用
6     // setMaxLifecycle 设置其最大生命周期为
7     Lifecycle.State.STARTED
8     if (fragment != mCurrentPrimaryItem) {
9         if (mCurrentPrimaryItem != null) {
10
11     mCurrentPrimaryItem.setMenuVisibility(false);
12         if (mBehavior ==
13 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
14             if (mCurTransaction == null) {
15                 mCurTransaction =
16 mFragmentManager.beginTransaction();
17             }
18
19 mCurTransaction.setMaxLifecycle(mCurrentPrimaryItem,
20 Lifecycle.State.STARTED);
21     } else {
22
23     mCurrentPrimaryItem.setUserVisibleHint(false);
24     }
25     }
26     //对于其他非可见的Fragment,则设置其最大生命周期为
27     //Lifecycle.State.RESUMED
28     fragment.setMenuVisibility(true);
29     if (mBehavior ==
30 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
31         if (mCurTransaction == null) {
32             mCurTransaction =
33 mFragmentManager.beginTransaction();
34     }
35 }
```

```
mCurTransaction.setMaxLifecycle(fragment,  
Lifecycle.State.RESUMED);  
    } else {  
        fragment.setUserVisibleHint(true);  
    }  
  
    mCurrentPrimaryItem = fragment;  
}  
}
```

既然在上述条件下，只有实际可见的 Fragment 会调用 onResume 方法，那是不是为我们提供了 ViewPager 下实现懒加载的新思路呢？也就是我们可以这样实现 Fragment 的懒加载：

```
1 abstract class LazyFragment : Fragment() {  
2  
3     private var isLoaded = false  
4  
5     override fun onResume() {  
6         super.onResume()  
7         if (!isLoaded) {  
8             lazyInit()  
9             Log.d(TAG, "lazyInit:!!!!!!")  
10            isLoaded = true  
11        }  
12    }  
13  
14    override fun onDestroyView() {  
15        super.onDestroyView()  
16        isLoaded = false  
17    }  
18  
19    abstract fun lazyInit()  
20 }
```

add+show+hide 模式下的新方案

虽然我们实现了Androidx 包下 ViewPager下的懒加载，但是我们仍然要考虑 add+show+hide 模式下的 Fragment 懒加载的情况，基于 ViewPager 在 `setPrimaryItem` 方法中的思路，我们可以在调用 add+show+hide 时，这样处理：

完整的代码请点击—>[ShowHideExt](#)

```
1  /**
2   * 使用add+show+hide模式加载fragment
3   *
4   * 默认显示位置[showPosition]的Fragment，最大
5   Lifecycle为Lifecycle.State.RESUMED
6   * 其他隐藏的Fragment，最大Lifecycle为
7   Lifecycle.State.STARTED
8   *
9   *@param containerviewId 容器id
10  *@param showPosition  fragments
11  *@param fragmentManager FragmentManager
12  *@param fragments 控制显示的Fragments
13  */
14 private fun loadFragmentsTransaction(
15     @IdRes containerviewId: Int,
16     showPosition: Int,
17     fragmentManager: FragmentManager,
18     vararg fragments: Fragment
19 ) {
20     if (fragments.isNotEmpty()) {
21
22     fragmentManager.beginTransaction().apply {
23         for (index in fragments.indices) {
24             val fragment = fragments[index]
25             add(containerviewId, fragment,
26                 fragment.javaClass.name)
27             if (showPosition == index) {
28                 setMaxLifecycle(fragment,
29                     Lifecycle.State.RESUMED)
29             } else {
30                 hide(fragment)
31                 setMaxLifecycle(fragment,
32                     Lifecycle.State.STARTED)
33             }
34         }
35     }
36 }
```

```
35     }
36
37     }.commit()
38 } else {
39     throw IllegalStateException(
40         "fragments must not empty"
41     )
42 }
43 }
44
45 /**
46  * 显示需要显示的Fragment[showFragment]，并设置其
47  * 最大Lifecycle为Lifecycle.State.RESUMED。
48  * 同时隐藏其他Fragment，并设置最大Lifecycle为
49  * Lifecycle.State.STARTED
50  * @param fragmentManager
51  * @param showFragment
52  */
53 private fun
54 showHideFragmentTransaction(fragmentManager:
55 FragmentManager, showFragment: Fragment) {
56     fragmentManager.beginTransaction().apply {
57         show(showFragment)
58         setMaxLifecycle(showFragment,
Lifecycle.State.RESUMED)

                //获取其中所有的fragment,其他的fragment进行
                隐藏
                val fragments =
fragmentManager.fragments
                for (fragment in fragments) {
                    if (fragment != showFragment) {
                        hide(fragment)
                        setMaxLifecycle(fragment,
Lifecycle.State.STARTED)
                    }
                }
            }
        }
    }
}
```

```
        }  
    }.commit()  
}
```

上述代码的实现也非常简单：

- 将需要显示的 Fragment，在调用 add 或 show 方法后，

```
setMaxLifecycle(showFragment, Lifecycle.State.RESUMED)
```

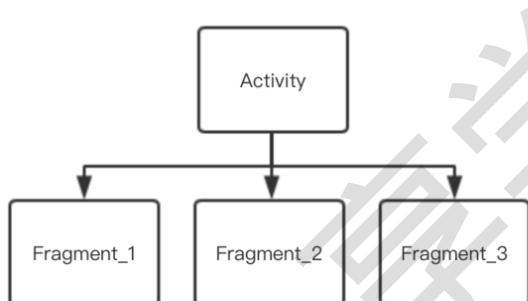
- 将需要隐藏的 Fragment，在调用 hide 方法后，

```
setMaxLifecycle(fragment, Lifecycle.State.STARTED)
```

结合上述操作模式，查看使用 setMaxLifecycle 后，Fragment 生命周期函数调用的情况。

add Fragment_1、Fragment_2、Fragment_3，并 hide Fragment_2,Fragment_3

:

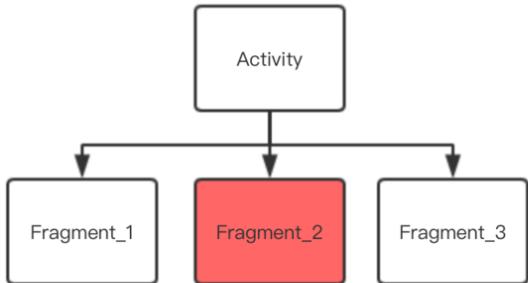


add Fragment_1...Fragment3
hide Fragment_2
hide Fragment_3

OneFragment: onAttach:
OneFragment: onCreate:
TwoFragment: onAttach:
TwoFragment: onCreate:
ThreeFragment: onAttach:
ThreeFragment: onCreate:
OneFragment: onViewCreated:
OneFragment: onActivityCreated:
TwoFragment: onViewCreated:
TwoFragment: onActivityCreated:
TwoFragment: onHiddenChanged:hidden-->true
ThreeFragment: onViewCreated:
ThreeFragment: onActivityCreated:
ThreeFragment: onHiddenChanged:hidden-->true
OneFragment: onStart:
TwoFragment: onStart:
ThreeFragment: onStart:
OneFragment: onResume:

生命周期函数调用日志

show Fragment_2, hide 其他 Fragment:

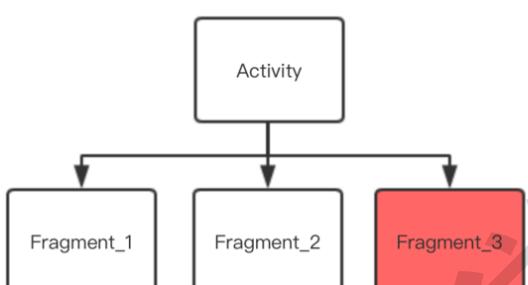


show Fragment_2
hide Fragment_1
hide Fragment_3

TwoFragment: onHiddenChanged:hidden-->false
TwoFragment: onResume:
OneFragment: onHiddenChanged:hidden-->true

生命周期函数调用日志

show Fragment_3 hide 其他 Fragment:



show Fragment_3
hide Fragment_1
hide Fragment_2

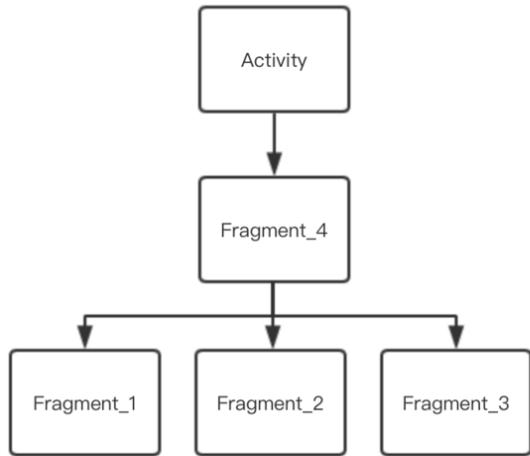
ThreeFragment: onHiddenChanged:hidden-->false
ThreeFragment: onResume:
TwoFragment: onHiddenChanged:hidden-->true

生命周期函数调用日志

参考上图，好像真的也能处理懒加载！！！！！美滋滋

并不完美的 setMaxLifecycle

当我第一次使用 `setMaxLifecycle` 方法时，我也和大家一样觉得万事大吉。但这套方案仍然有点点瑕疵，当 Fragment 的嵌套时，即使使用了 `setMaxLifecycle` 方法，第一次初始化时，同级不可见的Fragment，仍然 TMD 要调用可见生命周期方法。看下面的例子：



FourFragment: onAttach:
FourFragment: onCreate:
FourFragment: onViewCreated:
FourFragment: onActivityCreated:
FourFragment: onStart:
FourFragment: onResume:
OneFragment: onAttach:
OneFragment: onCreate:
TwoFragment: onAttach:
TwoFragment: onCreate:
ThreeFragment: onAttach:
ThreeFragment: onCreate:
OneFragment: onViewCreated:
OneFragment: onActivityCreated:
OneFragment: onStart:
OneFragment: onResume:
TwoFragment: onViewCreated:
TwoFragment: onActivityCreated:
TwoFragment: onStart:
TwoFragment: onResume:
TwoFragment: onHiddenChanged:hidden-->true
ThreeFragment: onViewCreated:
ThreeFragment: onActivityCreated:
ThreeFragment: onStart:
ThreeFragment: onResume:
ThreeFragment: onHiddenChanged:hidden-->true

生命周期函数调用日志

不知道是否是谷歌大大没有考虑到 Fragment 嵌套的情况，所以这里我们要对之前的方案就行修改，也就是如下所示：

```
1 abstract class LazyFragment : Fragment() {  
2  
3     private var isLoaded = false  
4  
5     override fun onResume() {  
6         super.onResume()  
7         //增加了Fragment是否可见的判断  
8         if (!isLoaded && !isHidden) {  
9             lazyInit()  
10            Log.d(TAG, "lazyInit!!!!!!")  
11            isLoaded = true  
12        }  
13    }  
14  
15    override fun onDestroyView() {  
16        super.onDestroyView()  
17        isLoaded = false  
18    }  
19  
20    abstract fun lazyInit()  
21  
22 }
```

在上述代码中，因为同级的 Fragment 在嵌套模式下，仍然要调用 onResume 方法，所以我们增加了 Fragment 可见性的判断，这样就能保证嵌套模式下，新方案也能完美的支持 Fragment 的懒加载。

ViewPager2 的处理方案

ViewPager2 本身就支持对实际可见的 Fragment 才调用 onResume 方法。关于 ViewPager2 的内部机制。感兴趣的小伙伴可以自行查看源码。

关于 ViewPager2 的懒加载测试，已上传至 [AndroidxLazyLoad](#)，大家可以结合项目查看Log日志。

两种方式的对比与总结

老一套的懒加载

- 优点：不用去控制 FragmentManager 的 add+show+hide 方法，所有的懒加载都是在 Fragment 内部控制，也就是控制 `setUserVisibleHint + onHiddenChanged` 这两个函数。
- 缺点：实际不可见的 Fragment，其 `onResume` 方法必然会被调用，这种反常规的逻辑，无法容忍。

新一套的懒加载（Androidx下setMaxLifecycle）

- 优点：在非特殊的情况下（缺点1），只有实际的可见 Fragment，其 `onResume` 方法才会被调用，这样才符合方法设计的初衷。
- 缺点：
 1. 对于 Fragment 的嵌套，及时使用了 `setMaxLifecycle` 方法。同级不可见的 Fragment，仍然要调用 `onResume` 方法。
 2. 需要在原有的 add+show+hide 方法中，继续调用 `setMaxLifecycle` 方法来控制 Fragment 的最大生命状态。

最后

这两种方案的优缺点已经非常明显了，到底该选择何种懒加载模式，还是要基于大家的意愿，作者我更倾向于使用新的方案。关于 Fragment 的懒加载实现，非常愿意听到大家不同的声音，如果你有更好的方案，可以在评论区留下您的 idea，期待您的回复。如果您觉得本篇文章对你有所帮助，请不要吝啬你的关注与点赞。ღ(' · υ ·)比心

2.4 Fragment全解析系列（一）：那些年踩过的坑

本篇主要介绍一些最常见的 Fragment 的坑以及官方 Fragment 库的那些自身的 BUG，并给出解决方案；这些 BUG 在你深度使用时会遇到，比如 Fragment 嵌套时或者单 Activity + 多 Fragment 架构时遇到的坑。

Fragment是可以让你的app纵享丝滑的设计，如果你的app想在现在基础上**性能大幅度提高，并且占用内存降低**，同样的界面Activity占用内存比Fragment要多，响应速度Fragment比Activity在中低端手机上快了很多，甚至能达到好几倍！如果你的app当前或以后有**移植**平板等平台时，可以让你节省大量时间和精力。

开始之前

最新版知乎，单Activity多Fragment的架构，响应可以说非常“丝滑”，非要说缺点的话，就是没有转场动画，并且转场会有类似闪屏现象。我猜测可能和Fragment转场动画的一些BUG有关。（这系列的最后一篇文章我会给出我的解决方案，可以自定义转场动画，并能在各种特殊情况下正常运行。）

但是！Fragment相比较Activity要难用很多，在多Fragment以及嵌套Fragment的情况下更是如此。

更重要的是Fragment的坑真的太多了，看Square公司的这篇文章吧，[Square：从今天开始抛弃Fragment吧！](#)

当然，不能说不再用Fragment，Fragment的这些坑都是有解决办法的，官方也在逐步修复一些BUG。

下面罗列一些，有常见的，也有极度隐蔽的一些坑，也是我在用单Activity多Fragment时遇到的坑，可能有更多坑可以挖掘...

在这之前为了方便后面文章的介绍，先规定一个“术语”，安卓app有一种特殊情况，就是app运行在后台的时候，系统资源紧张的时候导致把app的资源全部回收（杀死app的进程），这时把app再从后台返回到前台时，app会重启。这种情况下简称为：“**内存重启**”。（屏幕旋转等配置变化也会造成当前Activity重启，本质与“内存重启”类似）

在系统要把app回收之前，系统会把Activity的状态保存下来，Activity的FragmentManager负责把Activity中的Fragment保存起来。在“内存重启”后，Activity的恢复是从栈顶逐步恢复，Fragment会在宿主Activity的onCreate方法调用后紧接着恢复（从onAttach生命周期开始）。

getActivity()空指针

可能你遇到过`getActivity()`返回`null`, 或者平时运行完好的代码, 在“内存重启”之后, 调用`getActivity()`的地方却返回`null`, 报了空指针异常。

大多数情况下的原因: 你在调用了`getActivity()`时, 当前的Fragment已经`onDetach()`了宿主Activity。

比如: 你在`pop`了Fragment之后, 该Fragment的异步任务仍然在执行, 并且在执行完成后调用了`getActivity()`方法, 这样就会空指针。

解决办法:

更“安全”的方法: (对于Fragment已经`onDetach`这种情况, 我们应该避免在这之后再去调用宿主Activity对象, 比如取消这些异步任务, 但我们的团队可能会有粗心大意的情况, 所以下面给出的这个方案会保证安全)

在Fragment基类里设置一个Activity `mActivity`的全局变量, 在`onAttach(Activity activity)`里赋值, 使用`mActivity`代替`getActivity()`, 保证Fragment即使在`onDetach`后, 仍持有Activity的引用 (有引起内存泄露的风险, 但是异步任务没停止的情况下, 本身就可能已内存泄漏, 相比Crash, 这种做法“安全”些), 即:

```
protected Activity mActivity;
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    this.mActivity = activity;
}

/**
 * 如果你用了support 23的库, 上面的方法会提示过时, 有强迫症的小伙伴, 可以用下面的方法代替
 */
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    this.mActivity = (Activity)context;
}
```

异常：Can not perform this action after onSaveInstanceState

有很多小伙伴遇到这个异常，这个异常产生的原因是：

在你离开当前Activity等情况下，系统会调用onSaveInstanceState()帮你保存当前Activity的状态、数据等，直到再回到该Activity之前（onResume()之前），你执行Fragment事务，就会抛出该异常！（一般是其他Activity的回调让当前页面执行事务的情况，会引发该问题）

解决方法：

- **1、该事务使用commitAllowingStateLoss()方法提交，但是有可能导致该次提交无效！（宿主Activity被强杀时）**

对于popBackStack()没有对应的popBackStackAllowingStateLoss()方法，所以可以在下次可见时提交事务，参考2

- **2、利用onActivityResult()/onNewIntent()，可以做到事务的完整性，不会丢失事务**

一个简单的示例代码：

```
// ReceiverActivity 或 其子Fragment:  
void start(){  
    startActivityForResult(new Intent(this,  
SenderActivity.class), 100);  
}  
  
@Override  
protected void onActivityResult(int requestCode, int  
resultCode, Intent data) {  
    super.onActivityResult(requestCode, resultCode,  
data);  
    if (requestCode == 100 && resultCode == 100) {  
        // 执行Fragment事务  
    }  
}
```

```
// SenderActivity 或 其子Fragment:  
void do() { // 操作ReceiverActivity(或其子Fragment) 执行事务  
    setResult(100);  
    finish();  
}
```

Fragment重叠异常-----正确使用hide、show的姿势

在类onCreate()的方法加载Fragment，并且没有判断

saveInstanceState==null或

if(findFragmentByTag(mFragmentTag) == null)，导致重复加载了同一个Fragment导致重叠。（PS：replace情况下，如果没有加入回退栈，则不判断也不会造成重叠，但建议还是统一判断下）

```
@Override  
protected void onCreate(@Nullable Bundle  
savedInstanceState) {  
    // 在页面重启时，Fragment会被保存恢复，而此时再加载Fragment会重复  
    // 加载，导致重叠；  
    if(savedInstanceState == null){  
        // 或者 if(findFragmentByTag(mFragmentTag) == null)  
        // 正常情况下去 加载根Fragment  
    }  
}
```

详细原因：[从源码角度分析，为什么会发生Fragment重叠？](#)

~如果你add()了几个Fragment，使用show()、hide()方法控制，比如微信、QQ的底部tab等情景，如果你什么都不做的话，在“内存重启”后回到前台，app的这几个Fragment界面会重叠。~

~原因是FragmentManager帮我们管理Fragment，当发生“内存重启”，他会从栈底向栈顶的顺序一次性恢复Fragment；

但是因为官方没有保存Fragment的mHidden属性，默認為false，即show状态，所以所有Fragment都是以show的形式恢复，我们看到了界面重叠。

(如果是 replace，恢复形式和Activity一致，只有当你pop之后上一个Fragment才开始重新恢复，所有使用 replace 不会造成重叠现象) ~

v4-24.0.0+ 开始，官方修复了上述 没有保存mHidden的问题，所以如果你在使用24.0.0+的v4包，下面分析的2个解决方案可以自行跳过...

这里给出2个解决方案：

1、是大家比较熟悉的 `findFragmentByTag`：

即在 `add()` 或者 `replace()` 时绑定一个tag，一般我们是用fragment的类名作为tag，然后在发生“内存重启”时，通过 `findFragmentByTag` 找到对应的Fragment，并 `hide()` 需要隐藏的fragment。

下面是个标准恢复写法：

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity);  
  
    TargetFragment targetFragment;  
    HideFragment hideFragment;  
  
    if (savedInstanceState != null) { // “内存重启”时调用  
        targetFragment =  
getSupportFragmentManager().findFragmentByTag(TargetFrag-  
ent.class.getName());  
        hideFragment =  
getSupportFragmentManager().findFragmentByTag(HideFragmen-  
t.class.getName());  
        // 解决重叠问题  
        getFragmentManager().beginTransaction()  
            .show(targetFragment)  
            .hide(hideFragment)  
            .commit();  
    } else{ // 正常时  
        targetFragment = TargetFragment.newInstance();  
        hideFragment = HideFragment.newInstance();  
    }  
}
```

```
getFragmentManager().beginTransaction()
    .add(R.id.container, targetFragment,
targetFragment.getClass().getName())

.add(R.id.container, hideFragment, hideFragment.getClass().getName())
    .hide(hideFragment)
    .commit();
}

}
```

如果你想恢复到用户离开时的那个Fragment的界面，你还需要在 `onSaveInstanceState(Bundle outState)` 里保存离开时的那个可见的 tag 或下标，在 `onCreate` “内存重启” 代码块中，取出 tag/ 下标，进行恢复。

** 2、我的解决方案，9行代码解决所有情况的 Fragment 重叠：[传送门](#)**

Fragment 嵌套的那些坑

其实一些小伙伴遇到的很多嵌套的坑，大部分都是由于对嵌套的栈视图产生混乱，只要理清栈视图关系，做好恢复相关工作以及正确选择是使用 `getFragmentManager()` 还是 `getChildFragmentManager()` 就可以避免这些问题。

这部分内容是我们感觉 Fragment 非常难用的一个点，我会在[下一篇](#)中，详细介绍使用 Fragment 嵌套的一些技巧，以及如何清晰分析各个层级的栈视图。

附： `startActivityForResult` 接收返回问题

在 support 23.2.0 以下的支持库中，对于在嵌套子 Fragment 的 `startActivityForResult()`，会发现无论如何都不能在 `onActivityResult()` 中接收到返回值，只有最顶层的父 Fragment 才能接收到，这是一个 support v4 库的一个 BUG，不过在前两天发布的

support 23.2.0库中，已经修复了该问题，嵌套的子Fragment也能正常接收到返回数据了！

未必靠谱的出栈方法remove()

如果你想让某一个Fragment出栈，使用 `remove()` 在加入回退栈时并不靠谱。

如果你在add的同时将Fragment加入回退栈：`addToBackStack(name)`的情况下，它并不能真正将Fragment从栈内移除，如果你在2秒后（确保Fragment事务已经完成）打印

`getSupportFragmentManager().getFragments()`，会发现该Fragment依然存在，并且依然可以返回到被remove的Fragment，而且是空白页面。

如果你没有将Fragment加入回退栈，`remove`方法可以正常出栈。

如果你加入了回退栈，`popBackStack()` 系列方法才能真正出栈，这也就引入下一个深坑，`popBackStack(String tag, int flags)`等系列方法的BUG。

多个Fragment同时出栈的深坑BUG

6月17日更新：在support-25.4.0版本，google意识到下面的问题，并修复了。如果你使用25.4.0及以上版本，下面的方法不要再使用，google移除了mAvailIndices属性

在Fragment库中如下4个方法是可能产生BUG的：

- 1、`popBackStack(String tag,int flags)`
- 2、`popBackStack(int id,int flags)`
- 3、`popBackStackImmediate(String tag,int flags)`
- 4、`popBackStackImmediate(int id,int flags)`

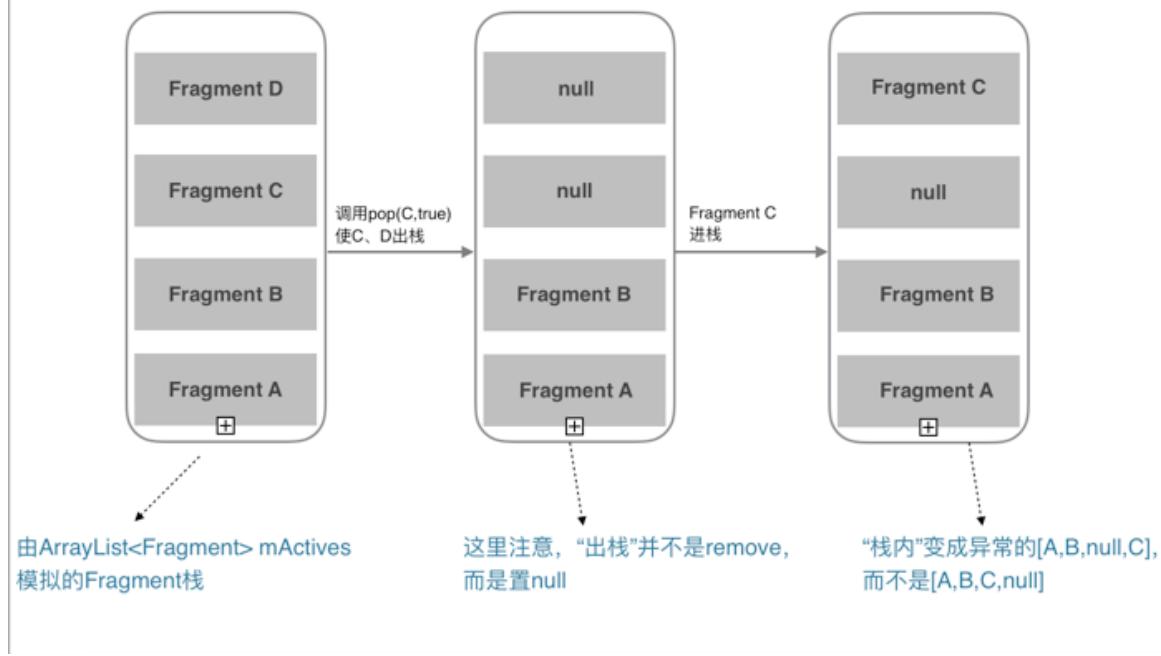
上面4个方法作用是，出栈到tag/id的fragment，即一次多个Fragment被出栈。

1、FragmentManager栈中管理fragment下标位置的数组ArrayList mAvailIndeices的BUG

下面的方法FragmentManagerImpl类方法，产生BUG的罪魁祸首是管理Fragment栈下标的mAvailIndices属性：

```
void makeActive(Fragment f) {  
    if (f.mIndex >= 0) {  
        return;  
    }  
    if (mAvailIndices == null || mAvailIndices.size()  
<= 0) {  
        if (mActive == null) {  
            mActive = new ArrayList<Fragment>();  
        }  
        f.setIndex(mActive.size(), mParent);  
        mActive.add(f);  
    } else {  
  
        f.setIndex(mAvailIndices.remove(mAvailIndices.size()-1),  
mParent);  
        mActive.set(f.mIndex, f);  
    }  
    if (DEBUG) Log.v(TAG, "Allocated fragment index " +  
f);  
}
```

上面代码最终导致了栈内顺序不正确的问题，如下图：



上面的这个情况，会一次异常，一次正常。带来的问题就是“内存重启”后，各种异常甚至Crash。

发现这BUG的时候，我一脸懵比，幸好，stackoverflow上有大神给出了[解决方案](#)！hack `FragmentManagerImpl`的`mAvailIndices`，对其进行一次`Collections.reverseOrder()`降序排序，保证栈内Fragment的index的正确。

```
public class FragmentTransactionBugFixHack {  
  
    public static void reorderIndices(FragmentManager  
fragmentManager) {  
        if (!(fragmentManager instanceof  
FragmentManagerImpl))  
            return;  
        FragmentManagerImpl fragmentManagerImpl =  
(FragmentManagerImpl) fragmentManager;  
        if (fragmentManagerImpl.mAvailIndices != null &&  
fragmentManagerImpl.mAvailIndices.size() > 1) {  
            Collections.sort(fragmentManagerImpl.mAvailIndices,  
Collections.reverseOrder());  
        }  
    }  
}
```

使用方法就是通过 `popBackStackImmediate(tag/id)` 多个 Fragment 后，调用

```
hanler.post(new Runnable(){  
    @Override  
    public void run() {  
  
FragmentTransactionBugFixHack.reorderIndices(fragmentMa  
ger));  
    }  
});
```

2、`popBackStack`的坑

`popBackStack` 和 `popBackStackImmediate` 的区别在于前者是加入到主线队列的末尾，等其它任务完成后才开始出栈，后者是队列内的任务立即执行，再将出栈任务放到队列尾（可以理解为立即出栈）。

如果你`popBackStack`多个Fragment后，紧接着`beginTransaction()``add`新的一个Fragment，接着发生了“内存重启”后，你再执行`popBackStack()`，app就会Crash，解决方案是`postDelay`出栈动画时间再执行其它事务，但是根据我的观察不是很稳定。

我的建议是：如果你想出栈多个Fragment，你应尽量使用`popBackStackImmediate(tag/id)`，而不是`popBackStack(tag/id)`，如果你想在出栈后，立刻`beginTransaction()`开始一项事务，你应该把事务的代码`post/postDelay`到主线程的消息队列里，下一篇有详细描述。

深坑 Fragment转场动画（仅分析v4包下的Fragment）

如果你的Fragment没有转场动画，或者使用`setCustomAnimations(enter, exit)`的话，那么上面的那些坑解决后，你可以愉快的玩耍了。

```
getFragmentManager().beginTransaction()
    .setCustomAnimations(enter, exit)
    // 如果你有通过tag/id同时出栈多个Fragment的情况时,
    // 请谨慎使用.setCustomAnimations(enter, exit,
    popEnter, popExit)
    // 在support-25.4.0之前出栈多Fragment时，伴随出栈动
    // 画，会在某些情况下发生异常
    // 你需要搭配Fragment的onCreateAnimation()临时取消出
    // 栈动画，或者延迟一个动画时间再执行一次上面提到的Hack方法，排序
```

(注意：如果你想给下一个Fragment设置进栈动画和出栈动画，`.setCustomAnimations(enter, exit)`只能设置进栈动画，第二个参数并不是设置出栈动画；
请使用`.setCustomAnimations(enter, exit, popEnter, popExit)`，这个方法的第一个参数对应进栈动画，第四个参数对应出栈动画，所以是`.setCustomAnimations(进栈动画, exit, popEnter, 出栈动画)`)

总结起来就是Fragment没有出栈动画的话，可以避免很多坑。
如果想让出栈动画运作正常的话，需要使用Fragment的`onCreateAnimation`中控制动画。

```
@Override  
public Animation onCreateAnimation(int transit, boolean  
enter, int nextAnim) {  
    // 此处设置动画  
}
```

但是用代价也是有的，你需要解决出栈动画带来的几个坑。

1、pop多个Fragment时转场动画带来的问题

6月17日更新：在support-25.4.0版本，google意识到下面动画引起的问题，并修复了。

在使用 `pop(tag/id)` 出栈多个Fragment的这种情况下，将转场动画临时取消或者延迟一个动画的时间再去执行其他事务；

原因在于这种情景下，可能会导致栈内顺序错乱（上文有提到），同时如果发生“内存重启”后，因为Fragment转场动画没结束时再执行其他方法，会导致Fragment状态不会被FragmentManager正常保存下来。

2、进入新的Fragment并立刻关闭当前Fragment时的一些问题

(1) 如果你想从当前Fragment进入一个新的Fragment，并且同时要关闭当前Fragment。由于数据结构是栈，所以正确做法是先 `pop`，再 `add`，但是转场动画会有覆盖的不正常现象，你需要特殊处理，不然会闪屏！

Tip:

如果你遇到Fragment的mNextAnim空指针的异常（通常是在你的Fragment被重启的情况下），那么你首先需要检查是否操作的Fragment是否为null；其次在你的Fragment转场动画还没结束时，你是否就执行了其他事务等方法；解决思路就是延迟一个动画时间再执行事务，或者临时将该Fragment设为无动画

总结

看了上面的介绍，你可能会觉得Fragment有点可怕。

但是我想说，如果你只是浅度使用，比如一个Activity容器包含列表Fragment + 详情Fragment这种简单情景下，不涉及到popBackStack/Immediate(tag/id)这些的方法，还是比较轻松使用的，出现的问题，网上都可以找到解决方案。

但是如果你的Fragment逻辑比较复杂，有特殊需求，或者你的app架构是仅有一个Activity + 多个Fragment，上面说的这些坑，你都应该全部解决。

在[下一篇](#)中，介绍了一些非常实用的使用技巧，包括如何解决Fragment嵌套、各种环境、组件下Fragment的使用等技巧，推荐阅读！

还有一些比较隐蔽的问题，不影响app的正常运行，仅仅是一些显示的BUG，并没有在上面介绍，在本系列的[最后一篇](#)，我给出了我的解决方案，一个我封装的[Fragmentation库](#)，解决了所有动画问题，非常适合**单Activity+多Fragment** 或者 **多模块Activity + 多Fragment**的架构。有兴趣的可以看看：)

2.5 Google-Fragment概览

2.6 Google-与其他Fragment通信

第三节 Service知识点

3.1 Handler知识点(必问)

Handler Looper Message 关系是什么？

Messagequeue 的数据结构是什么？为什么要用这个数据结构？

如何在子线程中创建 Handler？

Handler post 方法原理？

Handler消息机制，postDelayed会造成线程阻塞吗？对内存有什么影响？

当访问大量数据出现线程阻塞用什么技术解决？

3.2 Android主线程阻塞处理及优化

3.2.1 前期基础知识储备

当一个应用程序启动之后，android系统会为这个应用程序创建一个主线程（Main Thread），它负责渲染视图，分发事件到响应监听器并执行，对界面进行轮询的监听。因此，一般也叫做“**UI线程**”（UI Thread）。

android系统不会给应用程序的多个元素组件建立多个线程来执行。一个视图（Activity）中的多个view组件运行在同一个UI线程当中。因此，多个view组件的监听器的执行可能会相互影响。

如果在UI线程中做一些**比较耗时**的操作，比如**访问网络或者数据库**，都可能阻塞UI线程，导致时间停止分发（包括绘制事件）。对于用户来说，应用看起来像是卡住了，更坏的情况是，如果UI线程阻塞时间太长（超过5秒），android系统会弹出ANR（application not responding）错误提示框。

代码如下：

```
btn.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        //设置其耗时操作
        try {
            Thread.sleep(5000);      //5秒- 为Activity不响应报ANR错
误的时间
        } catch (InterruptedException e) {
            Log.i("线程沉睡",e.getMessage());
        }
        int sum=10;
        btn.setText("计数:"+sum);
    }
});
```

点击按钮之后，会发现界面卡死，这时就出现了**UI阻塞**。

3.2.2Android中如何处理耗时操作

上官方文档：

There are basically **two main ways** of having a Thread execute application code. One is providing a new class that extends Thread and overriding its run() method. The other is providing a new Thread instance with a Runnable object during its creation. In both cases, **the start() method** must be called to actually execute the new Thread.

官方文档中对于耗时操作提出的两点必须遵守的开发规则：

- ①**不要阻塞UI线程—即不要再UI线程中执行耗时操作；**
- ②**不要在UI线程之外的其他线程中，对视图中的组件进行设置。**

上述代码正确的写法：

```
btn.setOnClickListener(new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //设置其耗时操作  
                try {  
                    Thread.sleep(5000);  
                } catch (InterruptedException e) {  
                    Log.i("耗时操作", e.getMessage());  
                }  
                int sum=10;  
                btn.setText("计数："+sum);  
            }  
        }).start();  
    }  
});
```

运行结果，依然报错，我们已经开启支线程处理耗时操作，符合了第一条原则，但是，第二条“任何和UI界面相关的操作都应该放在主线程中完成”没有遵守。

Android对于这种情况，提供了**两种方法**，从支线任务回到主线程：

1) 通过View.post(Runnable)方法:

```
view.post(new Runnable() { //使用view.post(Runnable)进行组件设置
    @Override
    public void run() {
        //在这里进行UI操作，将结果显示在界面上
        btn.setText("计数："+sum)
    }
});
```

2) 通过runOnUiThread(Runnable)方法。

```
runOnUiThread (new Runnable(){
    @Override
    public void run() {
        //在这里进行UI操作，将结果显示在界面上
        btn.setText("计数："+sum)
    }
});
```

三种切回主线程的实例：

```
final Handler handler = new Handler();
new Thread(new Runnable() {
    @Override
    public void run() {
        // 素描算法处理 耗时操作
        final Bitmap bitmap1 =
sketchutil.testGaussBlur(finalBitmap,1,1);
        final Bitmap bitmap2 =
sketchutil.testGaussBlur(finalBitmap,10,10);
        final Bitmap bitmap3 =
sketchutil.testGaussBlur(finalBitmap,20,20);

        // 三种切回主线程更新UI的方法
        imageView.post(new Runnable() {
            @Override
            public void run() {
```

```
imageview.setImageBitmap(bitmap1); // 素描图
    }
});

runOnUiThread(new Runnable() {
    @Override
    public void run() {

originview.setImageBitmap(bitmap2); // 素描图
    }
});

handler.post(new Runnable() {
    @Override
    public void run() {

threeview.setImageBitmap(bitmap3); // 素描图
    }
});
}).start();
```

这种解决方法中，UI线程与我们新建的线程之间的关系类似于生产者与消费者之间的关系，新线程通过View.post(Runnable)和runOnUiThread(Runnable)方法在任务队列中加入任务，而UI线程对**任务队列进行轮询**，有任务的话就拿出来执行，修改界面。

但这种解决方法可读性和维护性较差，适用于**切换线程较少的场景**。

注意：使用handler方法切回主线程时，注意handler的实例化要放在主线程中，而不能在新开的子线程中，否则报错：

RuntimeException: Can't create handler inside thread that has not called Looper.prepare()

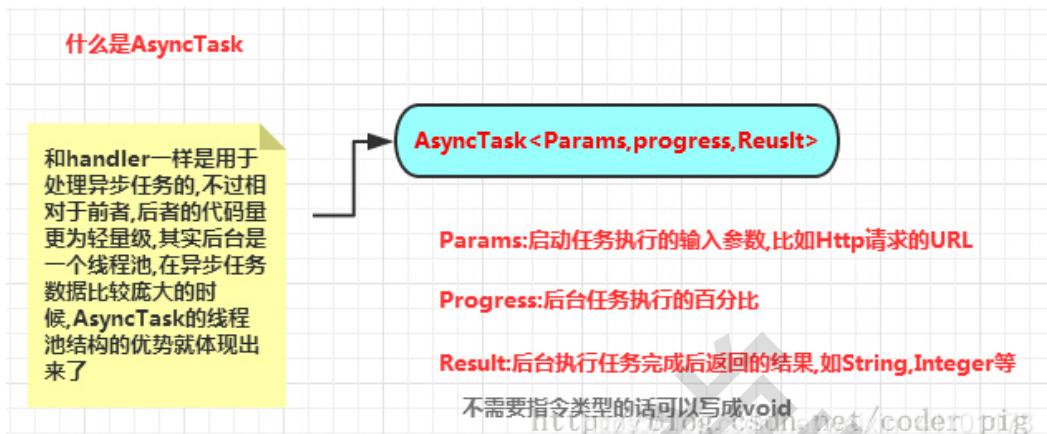
这是因为，Handler在哪里创建，就获得哪里的Looper。主线程创建的Handler，即默认使用主线程的Looper。

3.2.3Android中执行耗时操作和返回主线程最好的实现方式

即为Android中实现异步任务的方式：

(1) **Handler类**，在android中负责发送和处理消息，通过它可以实现其他支线线程与主线程之间的消息通讯。

(2) **AsyncTask类**，Android从1.5版本之后引入，使用它就可以非常灵活方便地从子线程切换到UI线程。



上代码，具体实现：

我是Handler类分隔线---

①不同的平台提供了不同的解决方案以实现跨线程跟新UI控件，Android为了解决这种问题引入了Handler机制。Handler类提供了两种方式解决我们在本文一开始遇到的问题（在一个新线程中更新主线程中的UI控件），一种是通过post方法，一种是调用sendMessage方法。

Post()方法：

```
class DownloadThread extends Thread{  
    @Override  
    public void run() {  
        try{  
            //此处让线程DownloadThread休眠5秒中，模拟文件的耗时过程  
            Thread.sleep(5000);  
            System.out.println("文件下载完成");  
            //文件下载完成后更新UI  
            Runnable runnable = new Runnable() {  
                @Override  
                public void run() {  
                    //更新UI操作  
                }  
            };  
            mHandler.post(runnable);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        MainActivity.this.statusTextView.setText("文件
下载完成");
    }
};

uiHandler.post(runnable); //post()在支线程中完成
UI操作
} catch (InterruptedException e) {
    e.printStackTrace();
}
};
```

sendMessage()方法：

```
class DownloadThread extends Thread {
    @Override
    public void run() {
        try {
            //此处让线程DownloadThread休眠5秒中，模拟文件
            的耗时过程
            Thread.sleep(5000);
            //文件下载完成后更新UI
            Message msg = new Message();
            msg.what = 1;
            msg.arg1 = 123;
            msg.arg2 = 321; //将该Message发送给对应的
Handler
            uiHandler.sendMessage(msg); //注意此时还是在
            支线程中
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    //uiHandler在主线程中创建，所以自动绑定主线程
    private Handler uiHandler = new Handler() {
        override
        public void handleMessage(Message msg) {
            switch (msg.what) {
```

```
        case 1:  
            System.out.println("msg.arg1:" +  
msg.arg1);  
            System.out.println("msg.arg2:" +  
msg.arg2);  
  
        MainActivity.this.statusTextview.setText("文件下载完成");  
        break;  
    }  
}  
};  
  
}
```

注意：sendMessage()方法中，设置**Message的what值**

Message.what是我们自定义的一个Message的识别码，以便于在Handler的handleMessage方法中根据what识别出不同的Message，以便我们做出不同的处理操作。设置Message的所携带的数据，简单数据可以通过两个int类型的field **arg1**和**arg2**来赋值，并可以在handleMessage中读取。如果Message需要携带复杂的数据，那么可以设置Message的**obj字段**，obj是Object类型，可以赋予任意类型的数据。

-----我是AsyncTask类分隔线---

② AsyncTask类，异步任务，从字面上说，就是我们的UI主线程运行的时候，异步的完成一些操作。AsyncTask允许我们执行一个异步任务在后台。我们可以将耗时操作放在异步任务当中执行，并随时将任务执行的结果返回给我们的UI主线程来更新我们的UI控件。通过ASyncTask我们就可以轻松的解决多线程之间的通信问题。

```
private class task extends AsyncTask<String, Void,  
String>{  
    @Override  
    protected String doInBackground(String... params) {  
        try {  
            Thread.sleep(5000); // 执行耗时操作  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }
    return "通过AsyncTask设置";
}
@Override
protected void onPostExecute(String result) {
    textView.setText(result);
}
};
```

在activity中建一个内部类，继承AsyncTask，重写doInBackground () 和onPostExecute () 方法，doInBackground的返回值会做为onPostExecute的参数执行。然后我们只需要在需要调用的时候new task().execute()就可以执行了。以下提供另一种写法，使用post方法切回主线程：

```
// 将Bitmap保存到本地
private void saveBitmapToOutput(Bitmap bitmap,
boolean isFilter) {
    AsyncTask.THREAD_POOL_EXECUTOR.execute(() -> {
        Uri output =
Uri.fromFile(generateImageFile());
        String outputPath = output.getPath();

        // 图片保存本地
        Bitmap.CompressFormat format =
Bitmap.CompressFormat.PNG;
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(new
File(outputPath));
            if (handlingview != null) {
                if (bitmap.compress(format, 100,
fos)) {
                    // 保存成功
                    handlingview.post(new Runnable()
{
                    @Override
                    public void run() {
```

```
        Log.d(TAG, "goToCropAct 图  
片抠图/应用滤镜成功 path = " + outputPath);  
        if (handlingView != null  
&& handlingView instanceof StickerView) {  
            mFilterPhotoBitmap =  
Bitmap.createBitmap(bitmap); /*抠图结果*/  
        }  
        if (!isFilter) {  
  
mEditFragment.setImageBitmap(bitmap); /*抠图成功后替换*/  
    } else {  
        /*若是滤镜/相框处理的结果  
则需要把原图设置过去*/  
  
mEditFragment.setImageBitmap(originBmp); /*将原始状态图片设置进去  
*/  
    }  
    ((StickerView)  
handlingView).replaceImageBitmap(bitmap, outputPath);  
    ((StickerView)  
handlingView).invalidate();  
}  
});  
} else {  
    // 保存失败  
    handlingView.post(new Runnable()  
{  
        @Override  
        public void run() {  
  
PrintUtils.showToast(PuzzleActivity.this,  
R.string.cut_pic_failed);  
    }  
});  
}  
}  
}  
} catch (Exception e) {
```

```
        Log.e(TAG, "goToCropAct : " +  
e.getMessage() );  
    } finally {  
        try {  
            if (fos != null) {  
                fos.close();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
});  
}  
  
private File generateImageFile() {  
    File file = getExternalCacheDir();  
    String fileName = "TEMP_IMG_" +  
System.currentTimeMillis() + ".jpg";  
    return new File(file, fileName);  
}
```

3.2.4两种异步更新UI方法的对比：

①Handler类：

优点：结构清晰，功能定义明确，对于后台**多个**任务时，简单清晰；

缺点：在单个后台异步处理时，显得代码过多，结构过于复杂（相对性）；

②AsyncTask类：

优点：简单快捷过程可控的轻量级异步类；

缺点：在使用多个异步操作和并需要UI变更时，就变得复杂起来；

总结：二者各有优劣，有各自对应的开发场景，**开发者都需要掌握。**

3.2.5附录，几个常用的有关Handler、Thread、Runnable类的操作

1) 执行延时操作 Handler + Runnable；

```
new Handler().postDelayed(new Runnable() {
    @Override
    public void run() {

        topBar.setBackgroundColor(getResources().getColor(R.color
                .translucent_bar_background));

        indicatorBar.setBackgroundColor(0x2059524D);

        findViewById(R.id.mask_view).setVisibility(View.GONE);
    }
}, 1000);
```

也可拆开实现

```
// 定义好Handler 和 Runnable接口
Handler handler = new Handler();
Runnable broad_thread = new Runnable() {
    public void run() {

        mOnScrollerViewUIChange.OnScrollerViewUIChange();
    }
};

// 调用 实现延时操作
//handler.removeCallbacks(broad_thread); 移除之前的
Runnable
handler.postDelayed(broad_thread, 1000);
```

2) 开启子线程 Thread + Runnable;

```
// Runnable 接口中只有run()方法，没有start()方法，所以记得加上start()
new Thread(new Runnable() {
    @Override
    public void run() {
        blur_bitmap = FastBlur.doBlur(scale_bitmap, 20,
false); // 高斯模糊 耗时操作
    }
}).start();
```

或者不选择使用内部类，而是直接在外部定义Runnable接口

```
// 定义外部类 实现Runnable接口
public class RestartCameraRunnable implements Runnable {
    @Override
    public void run() {
        glRender.addPreDrawTask(new Runnable() {
            @Override
            public void run() {
                cameraView.restartCamera(); // 重启相机
            }
        });
    }
}

// 使用 开启子线程
new Thread(new RestartCameraRunnable()).start();
```

注：尽量少在正式项目中使用类似于 new Thread 这种方式去开启子线程
(每次new Thread新建对象性能差，线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom)，建议根据实际情况决定：

①如果开启子线程次数有限，不频繁，就使用轻量级的 **HandlerThread** 实现消息处理操作和与主线程通信；

②如果开启子线程比较频繁，此时建议使用 **Java线程池**（Android线程概念源至Java）去处理；

3) 使用HandlerThread实现消息处理操作和与主线程通信；

```
public class MainActivity extends AppCompatActivity {

    Handler mainHandler,workHandler;
    HandlerThread mHandlerThread;
    TextView text;
    Button button1,button2;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = (TextView) findViewById(R.id.text1);

        // 创建与主线程关联的Handler
        mainHandler = new Handler();
        /**
         * 步骤1：创建HandlerThread实例对象
         * 传入参数 = 线程名字，作用 = 标记该线程
         */
        mHandlerThread = new
        HandlerThread("handlerThread");

        //步骤2：启动线程
        mHandlerThread.start();
        /**
         * 步骤3：创建工作线程Handler & 复写handleMessage()
         * 作用：关联HandlerThread的Looper对象、实现消息处理操作 & 与其他线程进行通信
         * 注：消息处理操作（HandlerMessage()）的执行线程 =
         * mHandlerThread所创建的工作线程中执行
         */
        workHandler = new
        Handler(mHandlerThread.getLooper()){
            @Override
```

```
public void handleMessage(Message msg)
{
    //通过msg来进行识别不同的操作 类似广播的过滤器
action 可扩展性非常强大
    switch(msg.what){
        case 1:
            try {
                //延时操作
                Thread.sleep(1000);
            } catch (InterruptedException e)
{
                e.printStackTrace();
            }
            // 通过主线程Handler.post方法进行在主
线程的UI更新操作
            mainHandler.post(new Runnable() {
                @Override
                public void run () {
                    text.setText("第一次执行");
                }
            });
            break;
        case 2:
            try {
                // 直接在handleMessage内部处理耗
时操作
                    Thread.sleep(3000);
            } catch (InterruptedException e)
{
                e.printStackTrace();
            }
            // 调用handler的post()方法处理UI操作
            mainHandler.post(new Runnable() {
                @Override
                public void run () {
                    text.setText("第二次执行");
                }
            });
    }
}
```

```
        break;
    default:
        break;
    }
}
};

/**
 * 步骤4：使用工作线程Handler向工作线程的消息队列发送消息
 * 在工作线程中，当消息循环时取出对应消息 & 在工作线程执行
相关操作
*/
button1 = (Button) findViewById(R.id.button1);
button1.setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Message msg = Message.obtain();
        msg.what = 1; //消息的标识
        msg.obj = "A"; // 消息的存放
        // 通过Handler发送消息到其绑定的消息队列
        workHandler.sendMessage(msg);
    }
});
button2 = (Button) findViewById(R.id.button2);
button2.setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Message msg = Message.obtain();
        msg.what = 2;
        msg.obj = "B";
        workHandler.sendMessage(msg);
    }
});
}

@Override
protected void onDestroy() {
```

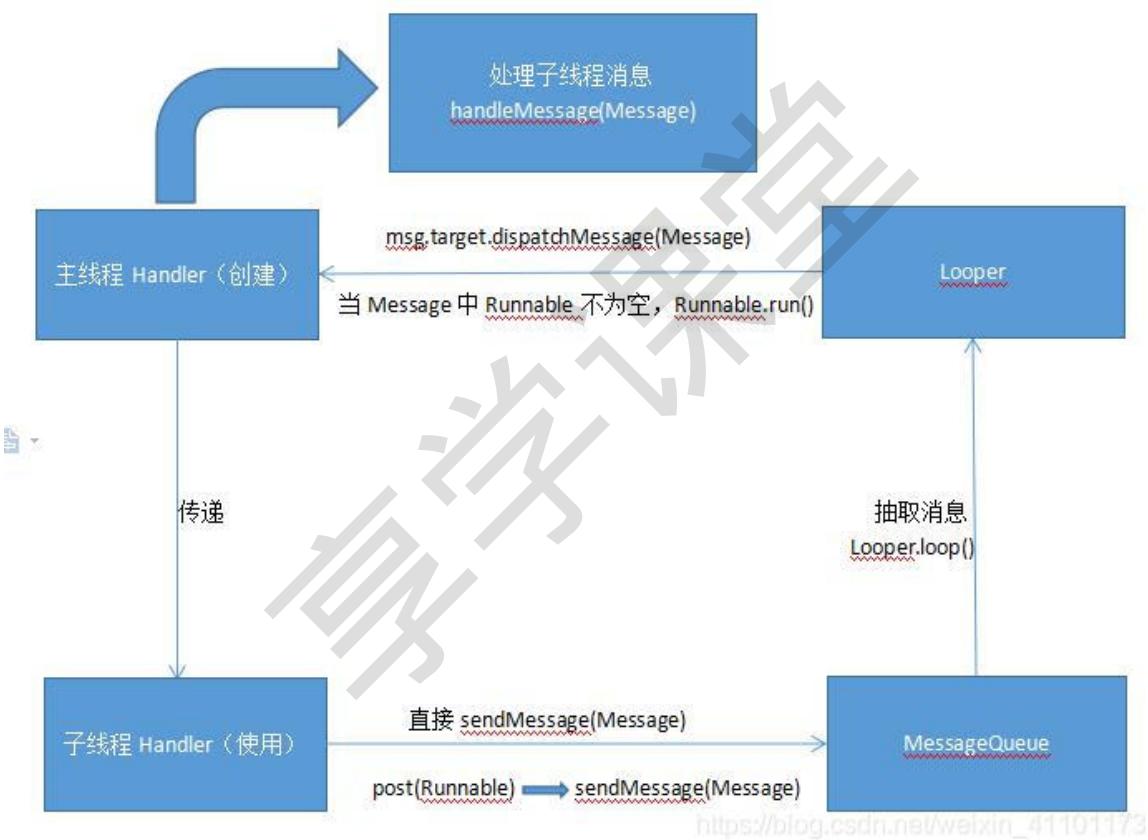
```

        super.onDestroy();
        mHandlerThread.quit(); // 退出消息循环
        workHandler.removeCallbacks(null); // 防止Handler
        内存泄露 清空消息队列
    }
}

```

HandlerThread在处理信息时，会根据what的值去做不同的操作，类似于动态注册的广播，所以在使用时十分方便。

提供一些其他Android消息机制分析，帮助理解读者理解：



①Handler是Android消息机制的上层接口，通过它可以轻松地将一个任务切换到Handler所在的线程中去执行，该线程既可以是主线程，也可以是子线程，要看构造Handler时使用的构造方法中传入的Looper位于哪里；

②Handler的运行需要底层的MessageQueue和Looper的支撑，**Handler创建的时候会采用当前线程的Looper来构造消息循环系统**，而线程默认是没有Looper的，如果需要使用Handler就必须为线程创建Looper；

③上述代码中的第一个Handler-mainHandler，实例化的时候，直接在onCreate()方法中new出了实例，其实是其已经在主线程中了，主线程-ActivityThread，**ActivityThread被创建时就会初始化Looper**，这就是主线程中默认可以直接使用Handler的原因；

④上述代码中的第二个Handler-workHandler，它在实例化的时候，参数传入了 **mHandlerThread.getLooper()**，注意，这个Handler使用的就不是主线程的Looper了，而是子线程的Looper，**HandlerThread在调用start()方法之后，就可以获取到子线程的Looper**，然后将其传入workHandler的构造方法中，那么此时的workHandler就会运行在子线程中，用于处理耗时操作。

⑤Handler的工作原理：Handler创建时会采用当前线程的Looper来构建内部消息循环系统，如果当前线程没有Looper，那么就会报错“Can't create handler inside thread that has not called Looper.prepare()”解决方法有两个：为当前线程创建Looper即可，像上述代码中workHandler，或者在一个有Looper的线程中创建Handler也行，就像上述代码中的mainHandler一样；

⑥调用Handler的**post**方法会将一个Runnable投递到Handler内部的Looper中去处理，也可以通过Handler的**send**方法来发送一个消息，这个消息同样会在Looper中去处理。其实**post**方法最终也是通过**send**方法来完成的。每当Looper发现有新消息到来时，就会处理这个消息，最终消息中的**Runnable的run方法**或者**Handler的handleMessage方法**就会被调用。注意Looper是运行在创建Handler所在的线程中的，这样一来Handler中的业务逻辑就被切换到创建Handler所在的线程中去执行了；

⑦Looper的工作原理：Looper在Android的消息机制中扮演着消息循环的角色，具体来说就是它会不停地从MessageQueue中查看是否有新消息，如果有新消息就会立刻处理，否则就一直阻塞在那里。注意关注一些重要的Looper的方法：

- Looper.prepare()-为当前线程创建一个Looper；
- Looper.loop()-开启消息循环，只有调用该方法，消息循环系统才会开始循环；
- Looper.prepareMainLooper()-为主线程也就是ActivityThread创建Looper使用；

- Looper.getMainLooper()-通过该方法可以在任意地方获取到主线程的Looper;
- Looper.quit() Looper.quitSafely()-退出Looper，自主创建的Looper建议在不使用的时候退出

⑧ActivityThread主线程通过ApplicationThread和AMS进行进程间通信

3.3 深入聊聊Android消息机制中的消息队列的设计

背景

Android系统中有一个核心的技术点——Android消息机制。

无论是系统开发者，还是应用开发者，深入理解这个机制，都是必要的，对日常的开发能起到事半功倍的效果。

在这个这个机制中，有四个非常重要的角色：Handler Looper MessageQueue Message，从它们的名字都能看出来它们各自的作用：

1. Handler：中文翻译叫手柄，可以用它控制消息的发送
2. Looper：一个循环辅助工具，可以让线程循环起来
3. MessageQueue：存储消息的数据结构，也就是接下来我将会深入分析的
4. Message：消息本身

作为系统的核心，高性能是必不可少的。这篇文章，侧重点主要在消息队列的设计上，但也避免不了涉及到其它的三个角色，因为它们是相互配合着的。

注意：下面涉及到的代码，是基于Android-28，老版本的代码可能会有细微差异，不过不影响阅读。

回顾下队列

我们都知道，在大部分有关数据结构的书籍中，Queue是一种先进先出的数据结构，和Stack的先进后出相对应。

队列的主要操作有：入队、出队、判断队列是否为空等。队列可以辅助解决一些广度优先遍历的问题，比如：二叉树的层序遍历

在日常开发中，Queue使用也还比较多的。例如：Volley源码中就使用了BlockingQueue维护Request；还有，OKHttp中同样是使用了Queue来维护Request

MessageQueue

Android系统中的这个消息队列，也符合传统队列的特性吗？它内部使用链表实现，还是数组实现？它也是先进先出的吗？

类似，但是又有点不一样。

链表 VS 数组？

队列可以用数组实现，也可以用链表实现，这要看具体的场景。

那Android消息机制中的这个MessageQueue应该选用什么数据结构来实现呢？

我们细想下，这里肯定选用链表，因为消息的插入和删除非常频繁，链表是较好的选择，而且，使用链表带来的另外一个好处，就是扩容很简单。

如果使用数组的话，扩容会涉及到数组的拷贝，这对性能损耗太大。

先进先出？

这点和传统的队列有点不一样，主要区别在于Android的这个队列中的消息是按照时间先后顺序来存储的，时间较早的消息，越靠近队头。

当然，我们也可以理解成，它是先进先出的，只是这里的先依据的不是谁先入队，而是消息待发送的时间。

如何实现唤醒和阻塞？

我们知道，Java中实现线程的唤醒和阻塞，有以下一些方法：

Object.wait() / notify()

Java并包中提供的一些工具类：CountDownLatch Barrier所使用的UnSafe底层的park方法

起初，Android系统的这个消息队列使用的是Object中的wait和notify；

后期，Google团队优化了下，使用了Linux系统的epoll机制，而且，此时的消息队列在Java层和Native层(c++层) 分别有一个，逻辑类似。

消息需要延时发送，怎么办？

当我们往消息队列中加入消息时，一般是通过Handler的一些方法，如下：

```
public final boolean post(Runnable r)
{
    return sendMessageDelayed(getPostMessage(r), 0);
}

public final boolean postAtTime(Runnable r, long uptimeMillis)
{
    return sendMessageAtTime(getPostMessage(r),
uptimeMillis);
}

.....



public final boolean sendMessageDelayed(Message msg,
long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg,
SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called
with no mQueue");

```

```
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}
```

最终都会调用sendMessageAtTime(Message msg, long uptimeMillis)。

看延迟发送的那个方法：sendMessageDelayed，我们会发现，它只是在将消息发送的时间加上需要延迟的间隔时间。

那最终又是如何实现延迟的呢？

一句话：利用的还是epoll机制。

epoll机制涉及到epoll_wait这个系统调用，有一个参数是timeout，表示epoll_wait()将要阻塞的毫秒数，具体描述如下：

The epoll_wait() system call waits for events on the epoll(7) instance referred to by the file descriptor epfd. The memory area pointed to by events will contain the events that will be available for the caller. Up to maxevents are returned by epoll_wait(). The maxevents argument must be greater than zero.

The timeout argument specifies the number of milliseconds that epoll_wait() will block. Time is measured against the CLOCK_MONOTONIC clock. The call will block until either:

- * a file descriptor delivers an event;
- * the call is interrupted by a signal handler; or
- * the timeout expires.

```
Note that the timeout interval will be rounded up to the
system clock
granularity, and kernel scheduling delays mean that the
blocking
interval may overrun by a small amount. Specifying a
timeout of -1
causes epoll_wait() to block indefinitely, while
specifying a timeout
equal to zero cause epoll_wait() to return immediately,
even if no
events are available.
```

所以，我们可以将消息的发送时间和系统当前时间作一个差值，将这个差值作为epoll_wait()的第三个参数timeout。

具体逻辑分布在如下这些代码中：

1. Java层的MessageQueue出队方法next()
2. Native层的[MessageQueue](#) pollOnce
3. Native层[Looper.cpp](#) pollOnce和pollInner方法

消息使用完了，就扔了吗？

答案肯定是否定的。

这里很有必要使用缓存技术，减少消息创建的成本。当我们每次需要创建Message的时候，从缓存池中获取，如果缓存池没有，再创建。

这就是为什么Handler中封装了一系列方法：obtainMessage

消息缓存池的逻辑如下：

```
 /**
 * Return a new Message instance from the global pool.
Allows us to
 * avoid allocating new objects in many cases.
*/
public static Message obtain() {
    synchronized (sPoolSync) {
```

```

        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null;
            m.flags = 0; // clear in-use flag
            sPoolSize--;
            return m;
        }
    }
    return new Message();
}

```

附上消息队列的入队和出队列方法源码

```

boolean enqueueMessage(Message msg, long when) {
    if (msg.target == null) {
        throw new IllegalArgumentException("Message
must have a target.");
    }
    if (msg.isInUse()) {
        throw new IllegalStateException(msg + " This
message is already in use.");
    }

    synchronized (this) {
        if (mQuitting) {
            IllegalStateException e = new
IllegalStateException(
                msg.target + " sending message to a
Handler on a dead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }

        msg.markInUse();
        msg.when = when;
        Message p = mMessages;

```

```

boolean needwake;
if (p == null || when == 0 || when < p.when) {
    // New head, wake up the event queue if
blocked.

    msg.next = p;
    mMessages = msg;
    needwake = mBlocked;
} else {
    // Inserted within the middle of the queue.
Usually we don't have to wake
    // up the event queue unless there is a
barrier at the head of the queue
    // and the message is the earliest
asynchronous message in the queue.
    needwake = mBlocked && p.target == null &&
msg.isAsynchronous();
    Message prev;
    for (;;) {
        prev = p;
        p = p.next;
        if (p == null || when < p.when) {
            break;
        }
        if (needwake && p.isAsynchronous()) {
            needwake = false;
        }
    }
    msg.next = p; // invariant: p == prev.next
    prev.next = msg;
}

// we can assume mPtr != 0 because mQuitting is
false.
if (needwake) {
    nativewake(mPtr);
}
return true;

```

```
}

Message next() {
    // Return here if the message loop has already
    // quit and been disposed.
    // This can happen if the application tries to
    // restart a looper after quit
    // which is not supported.
    final long ptr = mPtr;
    if (ptr == 0) {
        return null;
    }

    int pendingIdleHandlerCount = -1; // -1 only
during first iteration
    int nextPollTimeoutMillis = 0;
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }

        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // Try to retrieve the next message.
Return if found.
            final long now =
SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                // Stalled by a barrier. Find the
next asynchronous message in the queue.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null &&
!msg.isAsynchronous());
            }
        }
    }
}
```

```

        }

        if (msg != null) {
            if (now < msg.when) {
                // Next message is not ready.

Set a timeout to wake up when it is ready.
                nextPollTimeoutMillis = (int)
Math.min(msg.when - now, Integer.MAX_VALUE);
            } else {
                // Got a message.
                mBlocked = false;
                if (prevMsg != null) {
                    prevMsg.next = msg.next;
                } else {
                    mMessages = msg.next;
                }
                msg.next = null;
                if (DEBUG) Log.v(TAG, "Returning
message: " + msg);

                msg.markInUse();
                return msg;
            }
        } else {
            // No more messages.
            nextPollTimeoutMillis = -1;
        }

        // Process the quit message now that all
pending messages have been handled.
        if (mQuitting) {
            dispose();
            return null;
        }

        // If first time idle, then get the
number of idlers to run.
        // Idle handles only run if the queue is
empty or if the first message
    }
}

```

```
// in the queue (possibly a barrier) is
due to be handled in the future.

    if (pendingIdleHandlerCount < 0
        && (mMessages == null || now <
mMessages.when)) {
        pendingIdleHandlerCount =
mIdleHandlers.size();
    }
    if (pendingIdleHandlerCount <= 0) {
        // No idle handlers to run. Loop and
wait some more.

        mBlocked = true;
        continue;
    }

    if (mPendingIdleHandlers == null) {
        mPendingIdleHandlers = new
IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
    }
    mPendingIdleHandlers =
mIdleHandlers.toArray(mPendingIdleHandlers);
}

// Run the idle handlers.
// We only ever reach this code block during
the first iteration.

for (int i = 0; i < pendingIdleHandlerCount;
i++) {
    final IdleHandler idler =
mPendingIdleHandlers[i];
    mPendingIdleHandlers[i] = null; // release the reference to the handler

    boolean keep = false;
    try {
        keep = idler.queueIdle();
    } catch (Throwable t) {
```

```
        Log.wtf(TAG, "IdleHandler threw  
exception", t);  
    }  
  
    if (!keep) {  
        synchronized (this) {  
            mIdleHandlers.remove(idler);  
        }  
    }  
  
    // Reset the idle handler count to 0 so we do  
    // not run them again.  
    pendingIdleHandlerCount = 0;  
  
    // While calling an idle handler, a new  
    // message could have been delivered  
    // so go back and look again for a pending  
    // message without waiting.  
    nextPollTimeoutMillis = 0;  
}  
}
```

总结

当分析完Android消息机制中的这个MessageQueue，我们能体会到，为了提高性能，MessageQueue使用到了很多优雅的技术点。

3.4 深入理解MessageQueue

Android 中有两个非常重要的知识点，分别是Binder机制和Handler机制。前者用于跨进程通讯，并且通过ServiceManager给上层应用提供了大量的服务，而后者用于进程内部通讯，以消息队列的形式驱动应用的运行。之前的文章已经多次分析了Binder相关的内容，复杂程度远高于Handler，之后还会继续分析Binder。说到Handler，做安卓开发的一定都不会陌生，一般用于切换线程。其涉及到的类还有Looper，

MessageQueue, Message 等。其中MessageQueue是事件驱动的基础，本文会重点分析MessageQueue，其他内容会简单带过，可以参考生产者-消费者模式。

从Handler的入口开始分析：

```
`Looper.prepare();`
```

1. 创建一个Looper，并且是线程私有的：`sThreadLocal.set(new Looper(quitAllowed));`
2. 初始化Handler：`mHandler = new Handler();`，在构造函数中会获取线程私有的Looper，如获取不到会报错。
3. 开启无限循环：`Looper.loop();`。

在loop方法中主要代码如下：

```
for (;;) {  
    Message msg = queue.next(); // might block  
    if (msg == null) {  
        // No message indicates that the message queue is  
        // quitting.  
        return;  
    }  
    msg.target.dispatchMessage(msg);  
    msg.recycleUnchecked();  
}
```

1. 从MessageQueue中获取待处理的Message（阻塞线程）
2. 交给与之关联的Handler处理
3. 回收Message，供Message.obtain()复用

其中msg中的target是在Handler发送消息的时候赋值的：

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
```

```
MessageQueue queue = mQueue;
if (queue == null) {
    RuntimeException e = new RuntimeException(this +
" sendMessageAtTime() called with no mQueue");
    return false;
}
return enqueueMessage(queue, msg, uptimeMillis);
}

private boolean enqueueMessage(MessageQueue queue,
Message msg, long uptimeMillis) {
msg.target = this;
if (mAsynchronous) {
    msg.setAsynchronous(true);
}
return queue.enqueueMessage(msg, uptimeMillis);
}
```

发送的消息最终入队列到了MessageQueue。

简单总结一下Handler消息机制的工作原理：

1. 创建与线程绑定的Looper，同时会创建一个与之关联的MessageQueue用于存放消息
2. 开启消息循环，从MessageQueue中获取待处理消息，若无消息会阻塞线程
3. 通过Handler发送消息，此时会将Message入队列到MessageQueue中，并且唤醒等待的Looper
4. Looper获取的消息会投递给对应的Handler处理

可以看到其中与MessageQueue相关的也就两个操作，一个是入队列（MessageQueue是链表结构），一个是出队列，这正是本文介绍的重点。

MessageQueue的创建：

```
MessageQueue(boolean quitAllowed) {
    mQuitAllowed = quitAllowed;
    mPtr = nativeInit();
}
```

nativeInit()方法实现为android_os_MessageQueue_nativeInit():

[android_os_MessageQueue.cpp]

```
static jlong android_os_MessageQueue_nativeInit(JNIEnv*  
env, jclass clazz) {  
    NativeMessageQueue* nativeMessageQueue = new  
    NativeMessageQueue();  
    if (!nativeMessageQueue) {  
        jniThrowRuntimeexception(env, "Unable to allocate  
        native queue");  
        return 0;  
    }  
    nativeMessageQueue->incStrong(env);  
    return reinterpret_cast(nativeMessageQueue);  
}
```

这里会创建一个native层的MessageQueue，并且将引用地址返回给Java层保存在mPtr变量中，通过这种方式将Java层的对象与Native层的对象关联在了一起。这种在Java层保存Native层对象引用地址来实现关联的方式，在Android源代码中会经常看到。

然后看一下Native层MessageQueue的构造方法：

```
NativeMessageQueue::NativeMessageQueue() :  
    mPollEnv(NULL), mPollObj(NULL),  
    mExceptionObj(NULL) {  
    mLooper = Looper::getForThread();  
    if (mLooper == NULL) {  
        mLooper = new Looper(false);  
        Looper::setForThread(mLooper);  
    }  
}
```

也创建了一个Looper，并且也是与线程绑定的，事实上这个Looper与Java层的Looper并没有多大关系，一个是处理Native层时间的，一个是处理Java层事件的。

Java层的Looper会通过调用MessageQueue的next方法获取下一个消

息，先看主要部分，后面省略了一部分IdleHandler的处理逻辑，用于空闲的时候处理不紧急事件用的，有兴趣的自行分析：

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) {
        return null;
    }
    int pendingIdleHandlerCount = -1; // -1 only
during first iteration
    int nextPollTimeoutMillis = 0;
    for (;;) {
        nativePollOnce(ptr, nextPollTimeoutMillis);
        synchronized (this) {
            // Try to retrieve the next message.
Return if found.
            final long now =
SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                // Stalled by a barrier. Find the
next asynchronous message in the queue.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null &&
!msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    // Next message is not ready.
Set a timeout to wake up when it is ready.
                    nextPollTimeoutMillis = (int)
Math.min(msg.when - now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                }
            }
        }
    }
}
```

```
        if (prevMsg != null) {
            prevMsg.next = msg.next;
        } else {
            mMessages = msg.next;
        }
        msg.next = null;
        if (DEBUG) Log.v(TAG, "Returning
message: " + msg);
        msg.markInUse();
        return msg;
    }
} else {
    // No more messages.
    nextPollTimeoutMillis = -1;
}
// Process the quit message now that all
pending messages have been handled.
if (mQuitting) {
    dispose();
    return null;
}
pendingIdleHandlerCount = 0;
nextPollTimeoutMillis = 0;
}
}
```

这里有必要提一下MessageQueue的数据结构，是一个单向链表，Message对象有个next字段保存列表中的下一个，MessageQueue中的mMessages保存链表的第一个元素。

循环体内首先调用nativePollOnce(ptr, nextPollTimeoutMillis)，这是一个native方法，实际作用就是通过Native层的MessageQueue阻塞nextPollTimeoutMillis毫秒的时间。

- 1.如果nextPollTimeoutMillis=-1，一直阻塞不会超时。
- 2.如果nextPollTimeoutMillis=0，不会阻塞，立即返回。
- 3.如果nextPollTimeoutMillis>0，最长阻塞nextPollTimeoutMillis毫秒(超时)，如果期间有程序唤醒会立即返回。

暂时知道这些就可以继续向下分析了，native方法后面会讲到。
如果msg.target为null，则找出第一个异步消息，什么时候msg.target是null呢？看下面代码：

```
private int postSyncBarrier(long when) {
    // Enqueue a new sync barrier token.
    // we don't need to wake the queue because the
    purpose of a barrier is to stall it.
    synchronized (this) {
        final int token = mNextBarrierToken++;
        final Message msg = Message.obtain();
        msg.markInUse();
        msg.when = when;
        msg.arg1 = token;

        Message prev = null;
        Message p = mMessages;
        if (when != 0) {
            while (p != null && p.when <= when) {
                prev = p;
                p = p.next;
            }
        }
        if (prev != null) { // invariant: p ==
prev.next
            msg.next = p;
            prev.next = msg;
        } else {
            msg.next = p;
            mMessages = msg;
        }
        return token;
    }
}
```

这个方法直接在MessageQueue中插入了一个Message，并且未设置target。它的作用是插入一个消息屏障，这个屏障之后的所有同步消息都不会被执行，即使时间已经到了也不会执行。

可以通过public void removeSyncBarrier(int token)来移除这个屏障，参数是post方法的返回值。

这些方法是隐藏的或者是私有的，具体应用场景可以查看ViewRootImpl中的void scheduleTraversals()方法，它在绘图之前会插入一个消息屏障，绘制之后移除。

回到之前的next方法，如果发现了一个消息屏障，会循环找出第一个异步消息（如果有异步消息的话），所有同步消息都将忽略（平常发送的一般都是同步消息），可以通过setAsynchronous(boolean async)设置为异步消息。

继续往下，如果有消息需要处理，先判断时间有没有到，如果没到的话设置一下阻塞时间nextPollTimeoutMillis，进入下次循环的时候会调用nativePollOnce(ptr, nextPollTimeoutMillis);阻塞；

否则把消息返回给调用者，并且设置mBlocked = false代表目前没有阻塞。

如果阻塞了有两种方式唤醒，一种是超时了，一种是被主动唤醒了。根据生产消费模式，生产者有产品的时候一般情况下会唤醒消费者。那么MessageQueue入队列的时候应该会去唤醒，下面看一下MessageQueue入队列的方法，截取了主要逻辑：

```
boolean enqueueMessage(Message msg, long when) {
    synchronized (this) {
        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needwake;
        if (p == null || when == 0 || when < p.when)
        {
            msg.next = p;
            mMessages = msg;
            needwake = mBlocked;
        } else {
            needwake = mBlocked && p.target == null
&& msg.isAsynchronous();
```

```

        Message prev;
        for (;;) {
            prev = p;
            p = p.next;
            if (p == null || when < p.when) {
                break;
            }
            if (needwake && p.isAsynchronous()) {
                needwake = false;
            }
        }
        msg.next = p; // invariant: p ==
prev.next
        prev.next = msg;
    }
    if (needwake) {
        nativewake(mPtr);
    }
}
return true;
}

```

上面的代码主要就是加入链表的时候按时间顺序从小到大排序，然后判断是否需要唤醒，如果需要唤醒则调用 `nativewake(mPtr);` 来唤醒之前等待的线程。

再总结一下MessageQueue获取消息和加入消息的逻辑：

获取消息：

- 1.首次进入循环nextPollTimeoutMillis=0，阻塞方法nativePollOnce(ptr, nextPollTimeoutMillis)会立即返回
- 2.读取列表中的消息，如果发现消息屏障，则跳过后面的同步消息，总之会通过当前时间，是否遇到屏障来返回符合条件的待处理消息
- 3.如果没有符合条件的消息，会处理一些不紧急的任务
(IdleHandler)，再次进入第一步

加入消息：

- 1.加入消息比较简单，按时间顺序插入到消息链表中，如果是第一个那么根据mBlocked判断是否需要唤醒线程，如果不是第一个一般情况下不需要唤醒（如果加入的消息是异步的需要另外判断）

到这里其实关于MessageQueue已经分析的差不多了，其中有两个native方法没有涉及到分别是nativePollOnce, nativeWake，其实之前结论已经给出了，两个方法都会传入mPtr,在native层对应的是NativeMessageQueue的引用地址。

感兴趣的可以继续往下看，先看一下nativePollOnce的实现：

[android_os_MessageQueue.cpp]

```
static void
android_os_MessageQueue_nativePollOnce(JNIEnv* env,
 jobject obj,
 jlong ptr, jint timeoutMillis) {
    NativeMessageQueue* nativeMessageQueue =
reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->pollOnce(env, obj,
timeoutMillis);
}
```

通过传进来的ptr获取NativeMessageQueue对象的指针，然后调用NativeMessageQueue对象的pollOnce方法：

[android_os_MessageQueue.cpp]

```
void NativeMessageQueue::pollOnce(JNIEnv* env, jobject
pollObj, int timeoutMillis) {
    mPollEnv = env;
    mPollObj = pollObj;
    mLooper->pollOnce(timeoutMillis);
    mPollObj = NULL;
    mPollEnv = NULL;
    if (mExceptionObj) {
        env->Throw(mExceptionObj);
        env->DeleteLocalRef(mExceptionObj);
        mExceptionObj = NULL;
    }
}
```

调用的是Looper的pollOnce方法，这个Native层的Looper是在初始化NativeMessageQueue的时候创建的。

[Looper.cpp]

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
    int result = 0;
    for (;;) {
        while (mResponseIndex < mResponses.size()) {
            const Response& response =
                mResponses.itemAt(mResponseIndex++); int ident =
            response.request.ident; if (ident >= 0) {
                int fd = response.request.fd;
                int events = response.events;
                void* data = response.request.data;
                if (outFd != NULL) *outFd = fd;
                if (outEvents != NULL) *outEvents =
events;
                if (outData != NULL) *outData = data;
                return ident;
            }
        }
        if (result != 0) {
            if (outFd != NULL) *outFd = 0;
            if (outEvents != NULL) *outEvents = 0;
            if (outData != NULL) *outData = NULL;
            return result;
        }
        result = pollInner(timeoutMillis);
    }
}
```

先是处理native层的Response，这个直接跳过，最终调用pollInner

```
int Looper::pollInner(int timeoutMillis) {
    // Adjust the timeout based on when the next message
    // is due.
```

```
    if (timeoutMillis != 0 && mNextMessageUptime !=  
        LLONG_MAX) {  
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);  
        int messageTimeoutMillis =  
            toMillisecondTimeoutDelay(now, mNextMessageUptime);  
        if (messageTimeoutMillis >= 0  
            && (timeoutMillis < 0 ||  
                messageTimeoutMillis < timeoutMillis)) {  
            timeoutMillis = messageTimeoutMillis;  
        }  
    }  
  
    // Poll.  
    int result = POLL_WAKE;  
    mResponses.clear();  
    mResponseIndex = 0;  
  
    // We are about to idle.  
    mPolling = true;  
  
    struct epoll_event eventItems[EPOLL_MAX_EVENTS];  
    int eventCount = epoll_wait(mEpollFd, eventItems,  
        EPOLL_MAX_EVENTS, timeoutMillis);  
  
    // No longer idling.  
    mPolling = false;  
    // Acquire lock.  
    mLock.lock();  
    // Rebuild epoll set if needed.  
    if (mEpollRebuildRequired) {  
        mEpollRebuildRequired = false;  
        rebuildEpollLocked();  
        goto Done;  
    }  
    // Check for poll error.  
    if (eventCount < 0) {  
        if (errno == EINTR) {  
            goto Done;  
        }  
    }  
    // Process responses.  
    for (int i = 0; i < eventCount; i++) {  
        const epoll_event& event = eventItems[i];  
        if (event.events & POLLIN) {  
            mResponses.push_back(event.data.fd);  
        }  
    }  
    mResponseIndex = 0;  
    mPolling = true;  
}
```

```
        }

        ALOGW("Poll failed with an unexpected error: %s",
strerror(errno));
        result = POLL_ERROR;
        goto Done;
    }

    // Check for poll timeout.
    if (eventCount == 0) {
        result = POLL_TIMEOUT;
        goto Done;
    }

    // Handle all events.
    for (int i = 0; i < eventCount; i++) { int fd =
eventItems[i].data.fd; uint32_t epollEvents =
eventItems[i].events; if (fd == mWakeEventFd) { if
(epollEvents & EPOLLIN) { awoken(); } else {
ALOGW("Ignoring unexpected epoll events 0x%x on wake
event fd.", epollEvents); } } else { ssize_t requestIndex
= mRequests.indexOfKey(fd); if (requestIndex >= 0) {
            int events = 0;
            if (epollEvents & EPOLLIN) events |=
EVENT_INPUT;
            if (epollEvents & EPOLLOUT) events |=
EVENT_OUTPUT;
            if (epollEvents & EPOLLERR) events |=
EVENT_ERROR;
            if (epollEvents & EPOLLHUP) events |=
EVENT_HANGUP;
            pushResponse(events,
mRequests.valueAt(requestIndex));
        } else {
            ALOGW("Ignoring unexpected epoll events
0x%x on fd %d that is "
                  "no longer registered.",
epollEvents, fd);
        }
    }
}
```

```
    }

Done: ;
    // Invoke pending message callbacks.
    mNextMessageUptime = LLONG_MAX;
    while (mMessageEnvelopes.size() != 0) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        const MessageEnvelope& messageEnvelope =
mMessageEnvelopes.itemAt(0);
        if (messageEnvelope.uptime <= now) {
            { // obtain handler
                sp<Handler> handler = messageEnvelope.handler;
                Message message =
messageEnvelope.message;
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();
                handler->handleMessage(message);
            } // release handler

            mLock.lock();
            mSendingMessage = false;
            result = POLL_CALLBACK;
        } else {
            mNextMessageUptime = messageEnvelope.uptime;
            break;
        }
    }
    // Release lock.
    mLock.unlock();
    // Invoke all response callbacks.
    for (size_t i = 0; i < mResponses.size(); i++) {
Response& response = mResponses.editItemAt(i); if
(response.request.ident == POLL_CALLBACK) { int fd =
response.request.fd; int events = response.events; void*
data = response.request.data; int callbackResult =
response.request.callback->handleEvent(fd, events, data);
if (callbackResult == 0) {
    removeFd(fd, response.request.seq);
}
    }
}
```

```
        }
        response.request.callback.clear();
        result = POLL_CALLBACK;
    }
}

return result;
}
```

这个方法有点长，首先会根据Native Message的信息计算此次需要等待的时间，再调用

```
int eventCount = epoll_wait(mEpollFd, eventItems,
EPOLL_MAX_EVENTS, timeoutMillis);
```

来等待事件发生，其中是epoll是Linux下多路复用IO接口select/poll的增强版本，具体可以自行查阅相关文章，查考：[Linux IO模式及 select、poll、epoll详解](#)

如果epoll_wait返回了，那么可能是出错返回，可能是超时返回，可能是有事件返回，如果是前两种情况跳转到Done处。

如果有事件发生，会判断事件是否是mWakeEventFd（唤醒的时候写入的文件）做不同处理。在Done处会处理Native事件，还有Response。

总结一下就是，Java层的阻塞是通过native层的epoll监听文件描述符的写入事件来实现的。

最后看一下nativeWake：

```
static void android_os_MessageQueue_nativewake(JNIEnv*
env, jclass clazz, jlong ptr) {
    NativeMessageQueue* nativeMessageQueue =
reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->wake();
}
```

和之前一样，也是通过long类型的ptr获取NativeMessageQueue对象的指针，再调用wake方法：

```
void NativeMessageQueue::wake() {
    mLooper->wake();
}
```

调用的也是Looper的方法：

```
void Looper::wake() {
    uint64_t inc = 1;
    ssize_t nwrite =
TEMP_FAILURE_RETRY(write(mWakeEventFd, &inc,
sizeof(uint64_t)));
    if (nwrite != sizeof(uint64_t)) {
        if (errno != EAGAIN) {
            ALOGW("Could not write wake signal: %s",
strerror(errno));
        }
    }
}
```

重点是`write(mWakeEventFd, &inc, sizeof(uint64_t))`,写入了一个1，这个时候epoll就能监听到事件，也就被唤醒了

3.5 你真的懂Handler.postDelayed()的原理吗？

阅读之前先问大家一个问题：Handler.postDelayed()是先delay一定的时间，然后再放入messageQueue中，还是先直接放入MessageQueue中，然后在里面wait delay的时间？为什么？如果你答不上来的话，那么此文值得你看看。

原文：

使用handler发送消息时有两种方式，`post(Runnable r)`和`post(Runnable r, long delayMillis)`都是将指定Runnable（包装成PostMessage）加入到MessageQueue中，然后Looper不断从MessageQueue中读取Message进行处理。

然而我在使用的时候就一直有一个疑问，类似Looper这种「轮询」的工作方式，如果在每次读取时判断时间，是无论如何都会有误差的。但是在测试中发现Delay的误差并没有大于我使用
`System.out.println(System.currentTimeMillis())`所产生的误差，几乎可以忽略不计，那么Android是怎么做到的呢？

Handler.postDelayed()的调用路径

一步一步跟一下`Handler.postDelayed()`的调用路径：

1. Handler.postDelayed(Runnable r, long delayMillis)
2. Handler.sendMessageDelayed(getPostMessage(r), delayMillis)
3. Handler.sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis)
4. Handler.enqueueMessage(queue, msg, uptimeMillis)
5. MessageQueue.enqueueMessage(msg, uptimeMillis)

最后发现Handler没有自己处理Delay，而是交给了MessageQueue处理，我们继续跟进去看看MessageQueue又做了什么：

```
msg.markInUse();
msg.when = when;
Message p = mMessages;
boolean needwake;
if (p == null || when == 0 || when < p.when) {
    // New head, wake up the event queue if blocked.
    msg.next = p;
    mMessages = msg;
    needwake = mBlocked;
} else {
    ...
}
```

MessageQueue中组织Message的结构就是一个简单的单向链表，只保存了链表头部的引用（果然只是个Queue啊）。在`enqueueMessage()`的时候把应该执行的时间（上面Handler调用路径的第三步延迟已经加上了现有时间，所以叫when）设置到msg里面，并没有进行处理……WTF？

继续跟进去看看Looper是怎么读取MessageQueue的，在loop()方法内：

```
for (;;) {
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is
        // quitting.
        return;
    }
    ...
}
```

原来调用的是MessageQueue.next()，还贴心地注释了这个方法可能会阻塞，点进去看看：

```
for (;;) {
    if (nextPollTimeoutMillis != 0) {
        Binder.flushPendingCommands();
    }

    nativePollOnce(ptr, nextPollTimeoutMillis);

    synchronized (this) {
        // Try to retrieve the next message. Return if found.
        final long now = SystemClock.uptimeMillis();
        Message prevMsg = null;
        Message msg = mMessages;
        if (msg != null && msg.target == null) {
            // Stalled by a barrier. Find the next asynchronous
            // message in the queue.
            do {
                prevMsg = msg;
                msg = msg.next;
            } while (msg != null && !msg.isAsynchronous());
        }
        if (msg != null) {
            if (now < msg.when) {
```

```
// Next message is not ready. Set a timeout to wake up
when it is ready.
nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
} else {
// Got a message.
mBlocked = false;
if (prevMsg != null) {
prevMsg.next = msg.next;
} else {
mMessages = msg.next;
}
msg.next = null;
if (DEBUG) Log.v(TAG, "Returning message: " + msg);
msg.markInUse();
return msg;
}
} else {
// No more messages.
nextPollTimeoutMillis = -1;
}
...
}
```

可以看到，在这个方法内，如果头部的这个Message是有延迟而且延迟时间没到的 (`now < msg.when`)，会计算一下时间（保存为变量 `nextPollTimeoutMillis`），然后在循环开始的时候判断如果这个Message有延迟，就调用 `nativePollOnce(ptr, nextPollTimeoutMillis)` 进行阻塞。`nativePollOnce()` 的作用类似与 `object.wait()`，只不过是使用了Native的方法对这个线程精确时间的唤醒。

精确延时的问题到这里就算是基本解决了，不过我又产生了一个新的疑问：如果Message会阻塞MessageQueue的话，那么先postDelay10秒一个Runnable A，消息队列会一直阻塞，然后我再post一个Runnable B，B岂不是会等A执行完了再执行？正常使用时显然不是这样的，那么问题出在哪呢？

再来一步一步顺一下Looper、Handler、MessageQueue的调用执行逻辑，重新看到MessageQueue.enqueueMessage()的时候发现，似乎刚才遗漏了什么东西：

```
msg.markInUse();
msg.when = when;
Message p = mMessages;
boolean needwake;
if (p == null || when == 0 || when < p.when) {
// New head, wake up the event queue if blocked.
msg.next = p;
mMessages = msg;
needwake = mBlocked;
} else {
...
}
...
// we can assume mPtr != 0 because mQuitting is false.
if (needwake) {
nativewake(mPtr);
}
```

这个needWake变量和nativewake()方法似乎是唤醒线程啊？继续看看mBlocked是什么：

```
Message next() {
for (;;) {
...
if (msg != null) {
...
} else {
// Got a message.
mBlocked = false;
...
}
...
}
...
}
```

```
if (pendingIdleHandlerCount <= 0) {  
    // No idle handlers to run. Loop and wait some more.  
    mBlocked = true;  
    continue;  
}  
...  
}
```

就是这里了，在`next()`方法内部，如果有阻塞（没有消息了或者只有Delay的消息），会把`mBlocked`这个变量标记为`true`，在下一个Message进队时会判断这个message的位置，如果在队首就会调用`nativewake()`方法唤醒线程！

现在整个调用流程就比较清晰了，以刚刚的问题为例：

1. `postDelayed()`一个10秒钟的Runnable A、消息进队，`MessageQueue`调用`nativePollOnce()`阻塞，`Looper`阻塞；
2. 紧接着`post()`一个Runnable B、消息进队，判断现在A时间还没到、正在阻塞，把B插入消息队列的头部（A的前面），然后调用`nativewake()`方法唤醒线程；
3. `MessageQueue.next()`方法被唤醒后，重新开始读取消息链表，第一个消息B无延时，直接返回给`Looper`；
4. `Looper`处理完这个消息再次调用`next()`方法，`MessageQueue`继续读取消息链表，第二个消息A还没到时间，计算一下剩余时间（假如还剩9秒）继续调用`nativePollOnce()`阻塞；
5. 直到阻塞时间到或者下一次有Message进队；

这样，基本上就能保证`Handler.postDelayed()`发布的消息能在相对精确的时间被传递给`Looper`进行处理而又不会阻塞队列了。

另外，这里在阅读原文的基础上添加一点思考内容：

`MessageQueue`会根据`post delay`的时间排序放入到链表中，链表头的时间小，尾部时间最大。因此能保证时间Delay最长的不会block住时间短的。当每次`post message`的时候会进入到`MessageQueue`的`next()`方法，会根据其`delay`时间和链表头的比较，如果更短则，放入链表头，并

且看时间是否有delay，如果有，则block，等待时间到来唤醒执行，否则将唤醒立即执行。

所以handler.postDelay并不是先等待一定的时间再放入到MessageQueue中，而是直接进入MessageQueue，以MessageQueue的时间顺序排列和唤醒的方式结合实现的。使用后者的方式，我认为是集中式的统一管理了所有message，而如果像前者的话，有多少个delay message，则需要起多少个定时器。前者由于有了排序，而且保存的每个message的执行时间，因此只需一个定时器按顺序next即可。

3.6 Handler.postDelayed()是如何精确延迟指定时间的

使用Handler.postDelayed()时的疑问

使用handler发送消息时有两种方式，post(Runnable r)和post(Runnable r, long delayMillis)都是将指定Runnable（包装成PostMessage）加入到MessageQueue中，然后Looper不断从MessageQueue中读取Message进行处理。

然而我在使用的时候就一直有一个疑问，类似Looper这种「轮询」的工作方式，如果在每次读取时判断时间，是无论如何都会有误差的。但是在测试中发现Delay的误差并没有大于我使用System.out.println(System.currentTimeMillis())所产生的误差，几乎可以忽略不计，那么Android是怎么做到的呢？

Handler.postDelayed()的调用路径

一步一步跟一下Handler.postDelayed()的调用路径：

1. Handler.postDelayed(Runnable r, long delayMillis)
2. Handler.sendMessageDelayed(getPostMessage(r), delayMillis)
3. Handler.sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis)
4. Handler.enqueueMessage(queue, msg, uptimeMillis)
5. MessageQueue.enqueueMessage(msg, uptimeMillis)

最后发现Handler没有自己处理Delay，而是交给了MessageQueue处理，我们继续跟进去看看MessageQueue又做了什么：

```
msg.markInUse();
msg.when = when;
Message p = mMessages;
boolean needwake;
if (p == null || when == 0 || when < p.when) {
    // New head, wake up the event queue if blocked.
    msg.next = p;
    mMessages = msg;
    needwake = mBlocked;
} else {
    ...
}
```

MessageQueue中组织Message的结构就是一个简单的单向链表，只保存了链表头部的引用（果然只是个Queue啊）。在enqueueMessage()的时候把应该执行的时间（上面Handler调用路径的第三步延迟已经加上了现有时间，所以叫when）设置到msg里面，并没有进行处理.....WTF？

继续跟进去看看Looper是怎么读取MessageQueue的，在loop()方法内：

```
for (;;) {
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is
        // quitting.
        return;
    }
    ...
}
```

原来调用的是MessageQueue.next(), 还贴心地注释了这个方法可能会阻塞，点进去看看：

```
for (;;) {
    if (nextPollTimeoutMillis != 0) {
```

```
        Binder.flushPendingCommands();  
    }  
  
    nativePollOnce(ptr, nextPollTimeoutMillis);  
  
    synchronized (this) {  
        // Try to retrieve the next message.  Return if  
        found.  
        final long now = SystemClock.uptimeMillis();  
        Message prevMsg = null;  
        Message msg = mMessages;  
        if (msg != null && msg.target == null) {  
            // Stalled by a barrier.  Find the next  
            asynchronous message in the queue.  
            do {  
                prevMsg = msg;  
                msg = msg.next;  
            } while (msg != null &&  
!msg.isAsynchronous());  
        }  
        if (msg != null) {  
            if (now < msg.when) {  
                // Next message is not ready.  Set a  
                timeout to wake up when it is ready.  
                nextPollTimeoutMillis = (int)  
Math.min(msg.when - now, Integer.MAX_VALUE);  
            } else {  
                // Got a message.  
                mBlocked = false;  
                if (prevMsg != null) {  
                    prevMsg.next = msg.next;  
                } else {  
                    mMessages = msg.next;  
                }  
                msg.next = null;  
                if (DEBUG) Log.v(TAG, "Returning message:  
" + msg);  
                msg.markInuse();  
            }  
        }  
    }  
}
```

```
        return msg;
    }
} else {
    // No more messages.
    nextPollTimeoutMillis = -1;
}
...
}
```

可以看到，在这个方法内，如果头部的这个Message是有延迟而且延迟时间没到的 (`now < msg.when`)，会计算一下时间（保存为变量 `nextPollTimeoutMillis`），然后在循环开始的时候判断如果这个Message有延迟，就调用`nativePollOnce(ptr, nextPollTimeoutMillis);`进行阻塞。`nativePollOnce()`的作用类似与`object.wait()`，只不过是使用了Native的方法对这个线程精确时间的唤醒。

精确延时的问题到这里就算是基本解决了，不过我又产生了一个新的疑问：如果Message会阻塞MessageQueue的话，那么先`postDelay10秒`一个Runnable A，消息队列会一直阻塞，然后我再`post`一个Runnable B，B岂不是会等A执行完了再执行？正常使用时显然不是这样的，那么问题出在哪呢？

再来一步一步顺一下Looper、Handler、MessageQueue的调用执行逻辑，重新看到`MessageQueue.enqueueMessage()`的时候发现，似乎刚才遗漏了什么东西：

```
msg.markInUse();
msg.when = when;
Message p = mMessages;
boolean needwake;
if (p == null || when == 0 || when < p.when) {
    // New head, wake up the event queue if blocked.
    msg.next = p;
    mMessages = msg;
    needwake = mBlocked;
} else {
    ...
}
```

```
}

...
// we can assume mPtr != 0 because mQuitting is false.
if (needwake) {
    nativewake(mPtr);
}
```

这个needWake变量和nativeWake()方法似乎是唤醒线程啊？继续看看mBlocked是什么：

```
Message next() {
    for (;;) {
        ...
        if (msg != null) {
            ...
        } else {
            // Got a message.
            mBlocked = false;
            ...
        }
        ...
    }
    ...
    if (pendingIdleHandlerCount <= 0) {
        // No idle handlers to run. Loop and wait some
more.
        mBlocked = true;
        continue;
    }
    ...
}
```

就是这里了，在next()方法内部，如果有阻塞（没有消息了或者只有Delay的消息），会把mBlocked这个变量标记为true，在下一个Message进队时会判断这个message的位置，如果在队首就会调用nativeWake()方法唤醒线程！

现在整个调用流程就比较清晰了，以刚刚的问题为例：

1. postDelay()一个10秒钟的Runnable A、消息进队，MessageQueue调用nativePollOnce()阻塞，Looper阻塞；
2. 紧接着post()一个Runnable B、消息进队，判断现在A时间还没到、正在阻塞，把B插入消息队列的头部（A的前面），然后调用nativeWake()方法唤醒线程；
3. MessageQueue.next()方法被唤醒后，重新开始读取消息链表，第一个消息B无延时，直接返回给Looper；
4. Looper处理完这个消息再次调用next()方法，MessageQueue继续读取消息链表，第二个消息A还没到时间，计算一下剩余时间（假如还剩9秒）继续调用nativePollOnce()阻塞；
5. 直到阻塞时间到或者下一次有Message进队；

这样，基本上就能保证Handler.postDelayed()发布的消息能在相对精确的时间被传递给Looper进行处理而又不会阻塞队列了。

另外，这里在阅读原文的基础上添加一点思考内容：

MessageQueue会根据post delay的时间排序放入到链表中，链表头的时间小，尾部时间最大。因此能保证时间Delay最长的不会block住时间短的。当每次post message的时候会进入到MessageQueue的next()方法，会根据其delay时间和链表头的比较，如果更短则，放入链表头，并且看时间是否有delay，如果有，则block，等待时间到来唤醒执行，否则将唤醒立即执行。

所以handler.postDelay并不是先等待一定的时间再放入到MessageQueue中，而是直接进入MessageQueue，以MessageQueue的时间顺序排列和唤醒的方式结合实现的。使用后者的方式，我认为是集中式的统一管理了所有message，而如果像前者的话，有多少个delay message，则需要起多少个定时器。前者由于有了排序，而且保存的每个message的执行时间，因此只需一个定时器按顺序next即可。

3.7 Handler延迟消息执行机制，会阻塞吗？

3.7.1 Handler调用过程简单说明

Handler在Android中使用频繁，主要用来进行线程间通信，子线程通过Handler可以操作UI。有send和post两种方法，send方法是在工作线程中处理完耗时操作后调用handler的sendMessage (message) 把message对象发送给主线程，在主线程中重写handlerMessage () 方法，判断接收到的消息进行更新UI的操作；而post方法传递的是一个Runnable对象，Runnable的run方法最终执行在主线程中。

无论用那种方式最终都是调用sendMessageAtTime：

```
public final boolean postDelayed(Runnable r, Object token, long delayMillis)
{
    return sendMessageDelayed(getPostMessage(r, token), delayMillis);
}

public final boolean post(Runnable r)
{
    return sendMessageDelayed(getPostMessage(r), 0);
}
```

Post方法最终调用还是send方法

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0);
}

public final boolean sendEmptyMessage(int what)
{
    return sendEmptyMessageDelayed(what, 0);
}
```

```
public final boolean sendEmptyMessageDelayed(int what,
long delayMillis) {
    Message msg = Message.obtain();
    msg.what = what;
    return sendMessageDelayed(msg, delayMillis);
}

public final boolean sendMessageDelayed(Message msg, long
delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg,
SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(Message msg, long
uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with
no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}

public final boolean sendMessageDelayed(Message msg, long
delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg,
SystemClock.uptimeMillis() + delayMillis);
```

```
}
```

SendMessageAtTime第二个参数是利用系统开机时间加上**delayMillis**后计算得到的时间。

3.7.2 延迟消息如何处理

Looper采用轮询的工作方式，发送信息之后，**message**加入到**MessageQueue**中，然后**Looper**不断从**MessageQueue**中读取**Message**进行处理，延迟消息现实中使用中没有多大误差，如何从**MessageQueue**中读取延迟**Message**进行处理。

Handler不处理消息，Message放入MessageQueue：

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with "
                + "no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}
private boolean enqueueMessage(MessageQueue queue,
    Message msg, long uptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

Handler没有自己处理消息，最终放入了MessageQueue 设置了 target， target用于标识来源于哪个Handler， 最终取出消息后调用特定 Handler。

MessageQueue中的处理：

```
boolean enqueueMessage(Message msg, long when) {  
    synchronized (this) {  
        msg.markInUse();  
        msg.when = when;  
        Message p = mMessages;  
        boolean needwake;  
        if (p == null || when == 0 || when < p.when) {  
  
            msg.next = p;  
            mMessages = msg;  
            needwake = mBlocked;  
        } else {  
  
            needwake = mBlocked && p.target == null &&  
msg.isAsynchronous();  
            Message prev;  
            for (;;) {  
                prev = p;  
                p = p.next;  
                if (p == null || when < p.when) {  
                    break;  
                }  
                if (needwake && p.isAsynchronous()) {  
                    needwake = false;  
                }  
            }  
            msg.next = p; // invariant: p == prev.next  
            prev.next = msg;  
        }  
  
        //最终是否唤醒  
        if (needwake) {
```

```
        nativeWake(mPtr);
    }
}

return true;
}
```

在后面next()方法内部，如果有阻塞（没有消息了或者只有Delay的消息），会把mBlocked这个变量标记为true（下面会说明next方法），在下一个Message进队时会判断这个message的位置，如果在队首并且时间满足条件，会调用nativeWake()方法唤醒线程！

消息如何读取：

Looper内循环读取Message，looper的loop方法：

```
for (;;) {
    Message msg = queue.next(); // might block
    if (msg == null) {
        // No message indicates that the message queue is
        // quitting.
        return;
    }
}
```

Queue.next方法可能被阻塞，所以最终调用还是queue的next方法。

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) {
        return null;
    }

    int pendingIdleHandlerCount = -1; // -1 only during
    first iteration
    int nextPollTimeoutMillis = 0;
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }
    }
}
```

```
    }

    nativePollOnce(ptr, nextPollTimeoutMillis);

    synchronized (this) {
        // Try to retrieve the next message.  Return
        if found.

            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMssages;
            if (msg != null && msg.target == null) {
                // Stalled by a barrier.  Find the next
                asynchronous message in the queue.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null &&
!msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    // Next message is not ready.  Set a
                    timeout to wake up when it is ready.
                    nextPollTimeoutMillis = (int)
Math.min(msg.when - now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMssages = msg.next;
                    }
                    msg.next = null;
                    if (DEBUG) Log.v(TAG, "Returning
message: " + msg);
                    msg.markInUse();
                    return msg;
                }
            }
        }
    }
}
```

```
        }

    } else {
        // No more messages.
        nextPollTimeoutMillis = -1;
    }

    // Process the quit message now that all pending messages have been handled.
    if (mQuitting) {
        dispose();
        return null;
    }

    // If first time idle, then get the number of idlers to run.
    // Idle handles only run if the queue is empty or if the first message // in the queue (possibly a barrier) is due to be handled in the future.
    if (pendingIdleHandlerCount < 0
        && (mMessages == null || now < mMessages.when)) {
        pendingIdleHandlerCount =
mIdleHandlers.size();
    }
    if (pendingIdleHandlerCount <= 0) {
        // No idle handlers to run. Loop and wait some more.
        mBlocked = true;
        continue;
    }

    if (mPendingIdleHandlers == null) {
        mPendingIdleHandlers = new
IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
    }
    mPendingIdleHandlers =
mIdleHandlers.toArray(mPendingIdleHandlers);
```

```
    }

    // Run the idle handlers.
    // we only ever reach this code block during the
    first iteration.
    for (int i = 0; i < pendingIdleHandlerCount; i++)
{
    final IdleHandler idler =
mPendingIdleHandlers[i];
    mPendingIdleHandlers[i] = null; // release
the reference to the handler

    boolean keep = false;
    try {
        keep = idler.queueIdle();
    } catch (Throwable t) {
        Log.wtf(TAG, "IdleHandler threw
exception", t);
    }

    if (!keep) {
        synchronized (this) {
            mIdleHandlers.remove(idler);
        }
    }
}

// Reset the idle handler count to 0 so we do not
run them again.
pendingIdleHandlerCount = 0;

// while calling an idle handler, a new message
could have been delivered
// so go back and look again for a pending
message without waiting.
nextPollTimeoutMillis = 0;
}
```

取出消息，如果Message不满足执行条件，时间没到`now < msg.when`，计算下一个延迟 `nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE)`，继续阻塞；在循环开始的时候判断如果这个 Message有延迟，就调用`nativePollOnce(ptr, nextPollTimeoutMillis)`进行阻塞，如果时间到了直接分发消息进行调用。

由于执行delay消息会阻塞，那么如果发送多个delay消息，是不是前一个消息阻塞执行完了，才会执行后一个，现实使用中肯定不是这样的。上面已经说了在enqueueMessage方法中，加入新的消息时会首先判断需不需要唤醒线程，这样线程就不会一直阻塞（加入新的消息，唤醒线程，就不需要等待上一个阻塞的消息）。

delay消息会一直阻塞线程，直到延迟走完，或者下一个消息到来。

第四节 Intent知识点

4.1 Android跨进程传递大内存数据

背景

在主进程Activity 中选择或者编辑一张背景图产生一个bitmap 对象，要传递给 B进程，因为要尽量保证清晰度，所以这个bitmap还有可能比较大，所以必然会涉及到跨进程传输大型bitmap 的问题。

有哪些方案

跨进程传递大图，我们能想到哪些方案呢？

文件写入磁盘

最容易想到的方案就是先给图片保存到文件，给路径跨进程传过去，对方再从文件给图片decode出来，这个方案可行的，但是不够高效。

走系统IPC方式

另一种方案就是通过跨进程通信的方式，就是不走文件，直接走内存，这个肯定会快不少。跨进程通信有哪些方式呢？

首先Binder性能是可以，用起来也方便，但是有大小的限制，传的数据量大了就会抛异常。Socket或者管道性能不太好，涉及到至少两次拷贝。共享内存性能还不错，可以考虑，关键看怎么实现。

我们来看，通过Binder传图，一个是通过Intent传图，还有一个可以通过Binder调用传图。这两个不是一回事？你可能会有疑问，那么我们具体来看下这两种有什么区别。

先看常规Intent传图，我们塞进去一个超大的图。

这个

```
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("bitmap", mBitmap);
startActivity(intent);
```

bitmap 是一张很大的图片，正常传大图 抛出如下异常：

```
2019-07-27 20:40:38.981 19460-
19460/io.github.brijoe.ipcbigbitmapdemo E/AndroidRuntime:
FATAL EXCEPTION: main
    Process: io.github.brijoe.ipcbigbitmapdemo, PID:
19460
    ...
    Caused by: android.os.TransactionTooLargeException:
data parcel size 14293996 bytes
    ...
```

那我们看下系统对这个异常是怎么解释的

理解TransactionTooLargeException

The Binder transaction failed because it was too large. During a remote procedure call, the arguments and the return value of the call are transferred as Parcel objects stored in the Binder transaction buffer. If the arguments or the return value are too large to fit in the transaction buffer, then the call will fail and

TransactionTooLargeException will be thrown. The Binder transaction buffer has a limited fixed size, currently 1Mb, which is shared by all transactions in progress for the process. Consequently this exception can be thrown when there are many transactions in progress even when most of the individual transactions are of moderate size.

上面这段来自官方文档的描述，大致有这么几层意思：

1. 此异常发生在Binder IPC调用过程中。
2. 客户端发送请求数据过大 服务端返回数据过大 都会触发此异常。
3. binder调用的缓冲buffer大小当前为1M ,由当前进程共享。因此，如果同时有多个调用，就算单个调用过程传输的数据不大，也有可能触发此异常。
4. 可以将数据打碎分片成小数据来避免此异常。

很显然 由于我们传递的bitmap过大，导致了缓冲区超限，所以触发了此异常。那我们有没有办法能规避这个异常呢？我们来继续看。

神奇的putBinder 方法

再看Binder调用传图，是往Intent里塞了个Binder对象，等到另一个组件启动之后，读出这个Binder对象，调用它的getBitmap函数拿到Bitmap。

注意下， putBinder 方式 只在Android 4.3 (api 18) 及以上才有，所以我们做下判断。

发送端：

```
private void ipcBitmapBinder() {
    Intent intent = new Intent(this,
SecondActivity.class);
    Bundle bundle = new Bundle();
    if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.JELLY_BEAN_MR2) {
        bundle.putBinder("bitmap", new
ImageBinder(mBitmap));
    }
    intent.putExtras(bundle);
    startActivity(intent);
}
```

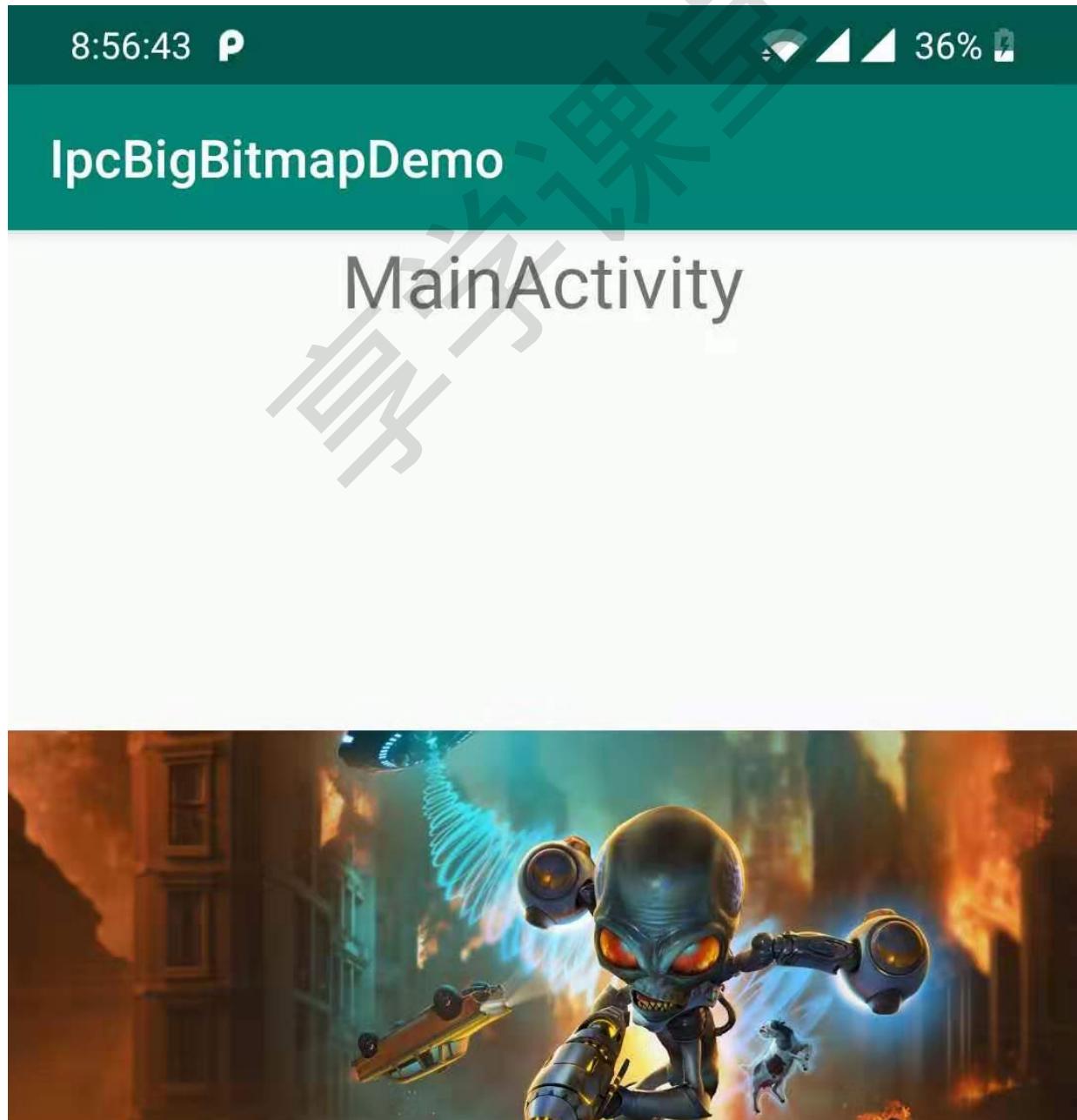
接收端

```
Bundle bundle = getIntent().getExtras();
if (bundle != null) {
    if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.JELLY_BEAN_MR2) {
        ImageBinder imageBinder = (ImageBinder)
bundle.getBinder("bitmap");
        Bitmap bitmap = imageBinder.getBitmap();
        mTv.setText(String.format(("bitmap大小
为%dkB"), bitmap.getByteCount() / 1024));
        mIv.setImageBitmap(bitmap);
    }
}
```

ImageBinder 是个继承自 Binder 的类，提供获取图片的方式。

```
class ImageBinder extends Binder {  
    private Bitmap bitmap;  
  
    public ImageBinder(Bitmap bitmap) {  
        this.bitmap = bitmap;  
    }  
  
    Bitmap getBitmap() {  
        return bitmap;  
    }  
}
```

点击启动按钮，Activity 正常跳转。我们将10几兆的图片通过跨进程的方式传过去了，避免了写磁盘的窘境。





bitmap大小为13958kB

INTENT传递

BINDER传递

8:57:21

36%

IpcBigBitmapDemo

SecondActivity



bitmap大小为13958kB

源码分析

这两个实现上有什么区别么？我们来看一下源码，就从startActivity开始吧，

```
int startActivity(..., Intent intent, ...) {  
    Parcel data = Parcel.obtain();  
    ....  
    intent.writeToParcel(data, 0);  
    ....  
    mRemote.transact(START_ACTIVITY_TRANSACTION, data,  
    reply, 0);  
    ....  
}
```

我们重点关注Bitmap是怎么传输的，这里给Intent写到Parcel了，通过下面这个writeToParcel函数，其实就是给Intent里的Bundle写到Parcel了，

```
public void writeToParcel(Parcel out, int flags) {  
    ....  
    out.writeBundle(mExtras);  
}
```

继续往下走，看Bundle怎么写到Parcel的，原来是调到了Bundle的writeToParcel函数，

```
public final void writeBundle(Bundle val) {  
    val.writeToParcel(this, 0);  
}
```

继续往下走，又调到了writeToParcelInner，

```
public void writeToParcel(Parcel parcel, int flags) {  
    final boolean oldAllowFds =  
    parcel.pushAllowFds(mAllowFds);  
    super.writeToParcelInner(parcel, flags);  
    parcel.restoreAllowFds(oldAllowFds);  
}
```

这个pushAllowFds是啥呢？就是说如果Bundle里不允许带描述符，那Bundle写到Parcel里的时候，Parcel也不许带描述符了。

```
bool Parcel::pushAllowFds(bool allowFds) {  
    const bool origValue = mAllowFds;  
    if (!allowFds) {  
        mAllowFds = false;  
    }  
    return origValue;  
}
```

我们再看writeToParcelInner函数，大家耐心一点，马上就要接近真相了，

```
void writeToParcelInner(Parcel parcel, int flags) {  
    .....  
    parcel.writeArrayMapInternal(mMap);  
}
```

这里调到了Parcel的writeArrayMapInternal函数，Bundle其实核心就是一个ArrayMap。写Bundle就是写这个ArrayMap。我们看这个Map是怎么写到Parcel的。

```
void writeArrayMapInternal(ArrayMap<String, Object> val)  
{  
    final int N = val.size();  
    writeInt(N);  
    for (int i = 0; i < N; i++) {  
        writeString(val.keyAt(i));  
       .writeValue(val.valueAt(i));  
    }  
}
```

这逻辑很简单啊，就在一个for循环里给map的key和value依次写到parcel。我们看writeValue是怎么写的，里面会根据value的不同类型采取不同的写法，

```
public final void writevalue(Object v) {  
    .....  
    else if (v instanceof Parcelable) {  
        writeInt(VAL_PARCELABLE);  
        writeParcelable((Parcelable) v, 0);  
    }  
    .....  
}
```

因为Bitmap是Parcelable的，所以我们只关注这个分支，这又调到了Bitmap的writeToParcel函数，

```
void writeToParcel.Parcelable p, int parcelableFlags) {  
  
    writeParcelableCreator(p);  
    p.writeToParcel(this, parcelableFlags);  
}
```

我们继续看Bitmap的writeToParcel，这又进入了native层，

```
public void writeToParcel(Parcel p, int flags) {  
    nativeWriteToParcel(mFinalizer.mNativeBitmap, ...);  
}
```

我们看native层是怎么实现的，

```
jboolean Bitmap_writeToParcel(JNIEnv* env, jobject, ...)  
{  
    android::Bitmap* androidBitmap =  
    reinterpret_cast<Bitmap*>(bitmapHandle);  
    androidBitmap->getSkBitmap(&bitmap);  
  
    // 往parcel里写Bitmap的各种配置参数  
  
    int fd = androidBitmap->getAshmemFd();  
    if (fd >= 0 && !isMutable && p->allowFds()) {
```

```

        status = p-
>writeDupImmutableBlobFileDescriptor(fd);
        return JNI_TRUE;
    }

    android::Parcel::WritableView blob;
    status = p->writeBlob(size, mutableCopy, &blob);
    const void* pSrc = bitmap.getPixels();
    memcpy(blob.data(), pSrc, size);
}

```

这里首先拿到native层的Bitmap对象，叫androidBitmap，然后拿到对应的SkBitmap。先看bitmap里带不带ashmemFd，如果带，并且这个Bitmap不能改，并且Parcel是允许带fd的话，就给fd写到parcel里，然后返回。否则的话继续往下，先有个WriteBlob对象，通过writeBlob函数给这个blob在parcel里分配了一块空间，然后给bitmap拷贝到这块空间里。我们看这个writeBlob函数，

```

status_t Parcel::writeBlob(size_t len, bool mutableCopy,
WritableView* outBlob) {
    if (!mAllowFds || len <= BLOB_INPLACE_LIMIT) {

        status = writeInt32(BLOB_INPLACE);
        void* ptr = writeInplace(len);
        outBlob->init(-1, ptr, len, false);
        return NO_ERROR;
    }
    int fd = ashmem_create_region("Parcel Blob", len);

    void* ptr = mmap(NULL, len, ..., MAP_SHARED, fd, 0);

    .....
    status = writeFileDescriptor(fd, true);
    outBlob->init(fd, ptr, len, mutableCopy);
    return NO_ERROR;
}

```

这个writeBlob函数，首先看如果不允许带fd，或者这个数据小于16K，就直接在parcel的缓冲区里分配一块空间来保存这个数据。不然的话呢，就另外开辟一个ashmem，映射出一块内存，数据就保存在ashmem的内存里，parcel里只写个fd就好了，这样就算数据量很大，parcel自己的缓冲区也不用很大。

为何putBinder 就不会抛异常？

bitmap的传输原理咱们清楚了，但是还有一个问题没有解决，为什么intent带大图会异常，但是binder调用带大图就没事呢？肯定是因为intent带bitmap的时候，bitmap直接拷到parcel缓冲区了，没有利用这个ashmem。为什么呢？

咱们注意到，只可能是这个allowFds没打开，咱们研究一下。

startActivity的时候会调到execStartActivity，这里会调到prepareToLeaveProcess，里面会禁用intent的allowFds，sendBroadcast也会这样，bindService也一样哈。

```
public ActivityResult execStartActivity(..., Intent
intent, ...) {
    .....
    intent.prepareToLeaveProcess();
    ActivityManagerNative.getDefault().startActivity(...);
}
```

至于为什么应用组件通信的时候要专门禁用这个fd，暂时没有找到特别可靠的解释，感觉应该是为了安全性考虑。

总结

Intent普通传大图方式为啥会抛异常而putBinder 为啥可以？
较大的bitmap直接通过Intent传递 是因为Intent启动组件时，系统禁掉了文件描述符fd，bitmap无法利用共享内存，只能采用拷贝到缓冲区的方式，导致缓冲区超限，触发异常；putBinder 的方式，避免了intent 禁用描

述符的影响，bitmap 写parcel时的fd 默认是true,可以利用到共享内存，所以能高效传输图片。

4.2数据存储

| 文件和数据库哪个效率高？

第三章 UI控件篇

第一节 屏幕适配

- dp+自适应布局+weight比例布局直接适配
- 今日头条适配方式
- 宽高限定符适配方式
- smallestWidth适配

1.1 Android屏幕适配和方案【整理】

前言

这里只是根据参考资料整理下，具体内容请阅读参考资料。

原型设计图

推荐1倍效果图，即采用 720 * 360 大小(1280 *720: 两倍图 \ 1920 * 1080: 三倍图)，最主要的原因就是1px = 1dp，效果图标多大的px，布局就写多大dp。

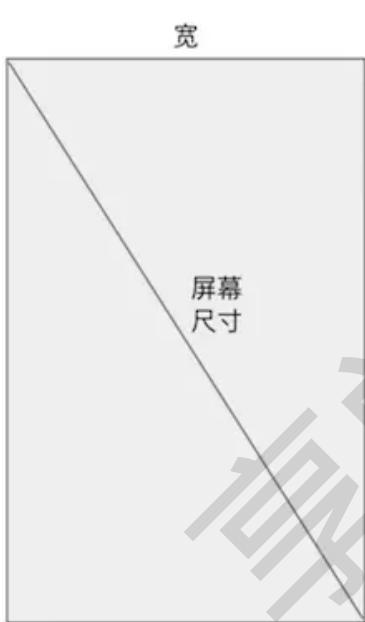
屏幕各项参数

- 手机像素 (px)：一个小黑点就是像素；
- 手机尺寸：屏幕的对角线的长度；

- 手机分辨率：整个屏幕一共有多少个点（像素），常见取值 480X800，320X480等；
- 像素密度 (dpi)：
 1. 每英寸中的像素数（假如设备分辨率为320*240，屏幕长2英寸宽1.5英寸， $dpi = 320/2 = 240/1.5 = 160$ ）
 2. 对应于DisplayMetrics类中属性densityDpi的值；
 3. 当然这种宽和高的dpi都相同的情况现在已经很少见，所以实际计算方式见下图；

像素密度(dpi)、屏幕尺寸、分辨率三者关系：

$$dpi = \frac{\sqrt{(宽^2 + 高^2)}(单位px)}{屏幕尺寸(单位inch)}$$



举例说明：屏幕分辨率为：1920*1080，屏幕尺寸为5吋的话，那么dpi为440：

- 密度 (density) ：
 1. 每平方英寸中的像素数 ($density = dpi / 160$) ；
 3. 对应于DisplayMetrics类中属性density的值；
 4. 可用于px与dp与dip的互相转换： $dp = px / density$ ；

| | | |
|-----------------|----------|-----------|
| ... | 1080*720 | 1920*1080 |
| dpi | 320 | 480 |
| dpi/160 density | 2 | 3 |

720P和1080P的手机，dpi是不同的，这也就意味着，不同的分辨率中，1dp对应不同数量的px(720P中，1dp=2px，1080P中1dp=3px)。

- 设备独立像素 (dp、dip) :

- 不同设备有不同的显示效果，不依赖像素 ($dp = px / density = px / (dpi / 160)$))
- dpi (像素密度) 为160 的设备上 $1dp = 1px$ 。

也叫dip(density independent pixel)直译为密度无关的像素。我们猜测如果使用了这个单位，我们在不同屏幕密度的设备上显示的长度就会是相同的。问题来了，在屏幕上显示图像时都是在屏幕上填充像素点，而使用这种与密度无关的像素 (我们在布局文件中使用的 dp/dip 就是与密度无关的像素) 是如何转换成像素的呢？

其实在安卓中，将屏幕密度为160dpi的中密度设备屏幕作为基准屏幕，在这个屏幕中， $1dp=1px$ 。其他屏幕密度的设备按照比例换算，具体如下表：

| 密度 | ldpi | mdpi | hdpi | xhdpi | xxhdpi | xxxhdpi |
|--------|-------|---------|---------|---------|---------|---------|
| dpi 范围 | 0–120 | 120–160 | 160–240 | 240–320 | 320–480 | 480–640 |
| 比例 | 0.75 | 1 | 1.5 | 2 | 3 | 4 |
| 整数比例 | 3 | 4 | 6 | 8 | 12 | 16 |

由上表不难计算1dp在hdpi设备下等于1.5px，同样的在xxhdpi设备下 $1dp=3px$ 。这里我们从dp到px解释了Android中不同屏幕密度之间的像素比例关系。

下面换一个角度，从px到dp的变化来说明这种比例关系。

就拿为App设计icon来说，为了让App的icon在不同的屏幕密度设备显示相同 (这里我们选择在以mipmap开头的目录中都要设计一个icon)，就必须要求icon在屏幕中占据相同的dp。那么对于不同的屏幕密度 (MDPI、HDPI、XHDPI、XXHDPI 和 XXXHDPI) 应按照 2:3:4:6:8 的比例进行缩放。比如说一个icon的尺寸为48x48dp，这表示在 MDPI 的屏幕

上其实际尺寸应为 48x48px，在 HDPI 的屏幕上其实际大小是 MDPI 的 1.5 倍 (72x72 px)，在 XDPI 的屏幕上其实际大小是 MDPI 的 2 倍 (96x96 px)，依此类推。

- 放大像素 (sp)：用于字体显示；
- dp转px、px转dp：

```
public class Dp2Px {  
    public static int dp2px(Context context, int dp) {  
        return (int) (dp *  
context.getResources().getDisplayMetrics().density +  
0.5);  
    }  
  
    public static int px2dp(Context context, int px) {  
        return (int) (px /  
context.getResources().getDisplayMetrics().density +  
0.5);  
    }  
}
```

说明：0.5 是为了避免损失精度。

适配分类

一、图片适配

在AndroidStudio的资源目录res下有五个层级图片文件夹，分别用来存放不同分辨率的图片：

- drawable-ldpi：低分辨率（用的少了，一般不再用）
- drawable-mdpi：中分辨率
- drawable-hdpi：高分辨率
- drawable-xdpi：较高分辨率
- drawable-xxdpi：超级高分辨率
- drawable-xxxhdpi：顶级分辨率

在对应的文件夹下放置不同分辨率的图片就可以很好的对图片进行适配。随着屏幕越来越大，推荐xxdpi的一套切图，这样就可以向下和向上兼容，节省资源。

建议图标使用svg格式，图片仍然使用png格式，svg的图标大小约是png的1/4，在很大的项目中，图标有很多，这个时候svg的优势就凸显无疑了。

二、布局适配

在layout之外，根据不同分辨率，创建不同的布局文件夹：

- layout-800 * 480
- layout-1280 * 720

手机会根据分辨率去找设定的不同大小的layout的布局。实际开发中这种使用的情况非常少，因为占用太多资源，慎用。

三、权重适配

当两个或者更多布局占满屏幕宽或高的时候，子布局可以使用权重适配，常见于LinearLayout线性布局中。

四、屏幕适配

详细内容请阅读参考资料。以下内容均是引用！

1、dp+自适应布局+weight比例布局直接适配

这基本是最原始的Android适配方案。

wrap_content,match_parent,layout_weight等，我们就要毫不犹豫的使用，而且在高这个维度上，我们要依照情况设计为可滑动的方式，或者match_parent尽量不要写死。总之，**所有的适配方案都不是用来取代match_parent,wrap_content的，而是用来完善他们的。**

缺点

(1) 这只能保证我们写出来的界面适配绝大部分手机，部分手机仍然需要单独适配；

为什么dp只解决了90%的适配问题，因为并不是所有的1080P的手机dpi都是480，比如Google 的Pixel2 (1920*1080) 的dpi是420，也就是说，在Pixel2中， $1\text{dp}=2.625\text{px}$,这样会导致相同分辨率的手机中，这样，一个100dp*100dp的控件，在一般的1080P手机上，可能都是300px,而Pixel 2 中，就只有262.5px,这样控件的实际大小会有所不同。

(2) 这种方式无法快速高效的把设计师的设计稿实现到布局代码中，通过dp直接适配，我们只能让UI基本适配不同的手机,但是在设计图和UI代码之间的鸿沟，dp是无法解决的，因为dp不是真实像素。而且，设计稿的宽高往往和Android的手机真实宽高差别极大，以我们的设计稿为例，设计稿的宽高是375px*750px，而真实手机可能普遍是1080*1920；那么在日常开发中我们是怎么跨过这个鸿沟的呢？基本都是通过百分比啊，或者通过估算，或者设定一个规范值等等。总之，当我们拿到设计稿的时候，设计稿的ImageView是128px*128px，当我们在编写layout文件的时候，却不能直接写成128dp*128dp。在把设计稿向UI代码转换的过程中，我们需要耗费相当的精力去转换尺寸，这会极大的降低我们的生产力，拉低开发效率。

2、dimens基于px的适配（宽高限定符适配）

原理

根据市面上手机分辨率的占比分析，我们选定一个占比例值大的（比如1280*720）设定为一个基准，然后其他分辨率根据这个基准做适配。

基准的意思（比如320*480的分辨率为基准）是：

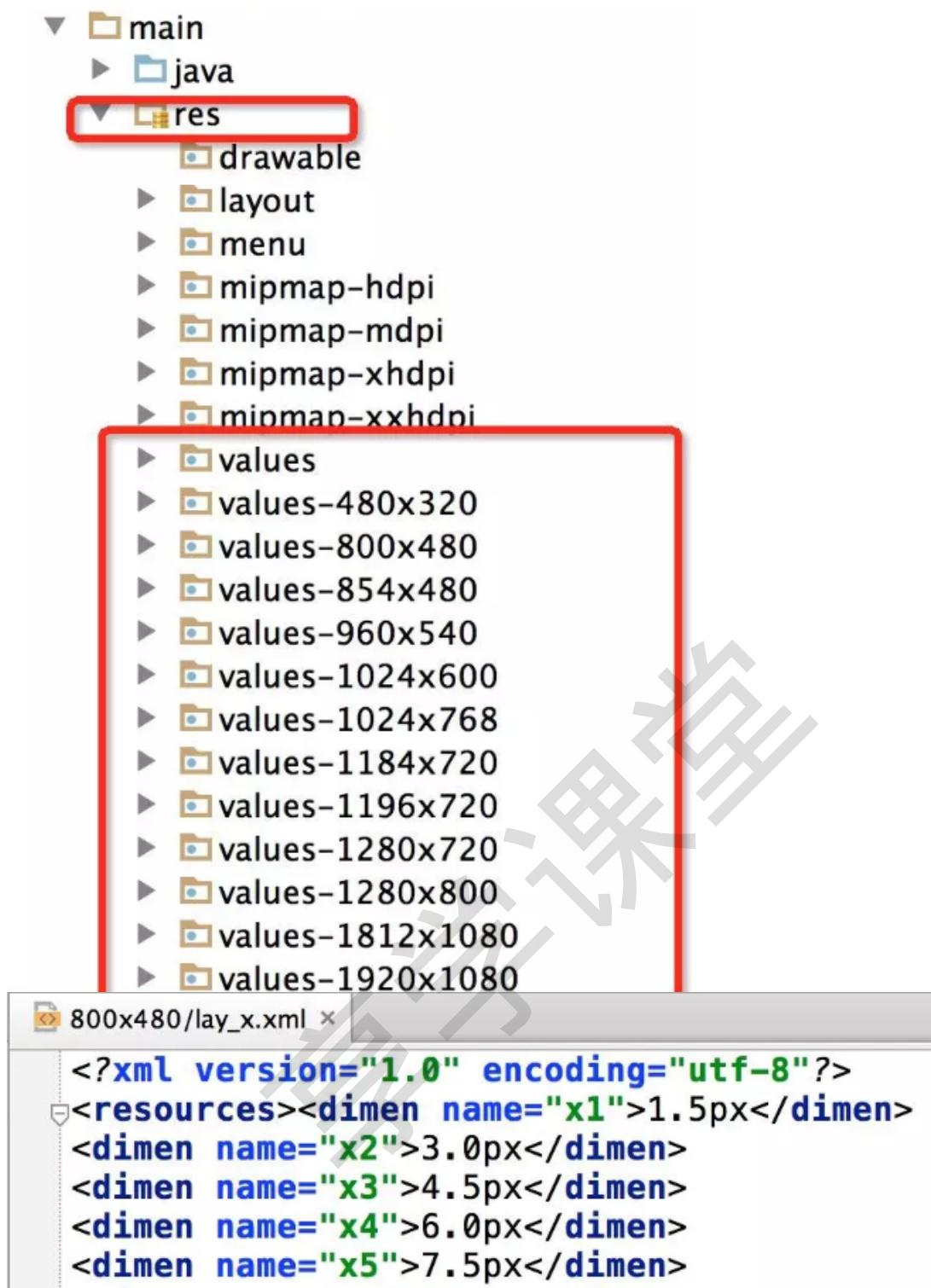
宽为320，将任何分辨率的宽度分为320份，取值为x1到x320

长为480，将任何分辨率的高度分为480份，取值为y1到y480

例如对于800 * 480的分辨率设备来讲，需要在项目中values-800x480目录下的dimens.xml文件中的如下设置（当然了，可以通过工具自动生成）：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<dimen name="x1">1.5px</dimen>
<dimen name="x2">3.0px</dimen>
<dimen name="x3">4.5px</dimen>
<dimen name="x4">6.0px</dimen>
<dimen name="x5">7.5px</dimen>
```

可以看到 $x1 = 480 / \text{基准} = 480 / 320 = 1.5$;它的意思就是同样的1px，在320/480分辨率的手机上是1px，在480/800的分辨率的手机上就是 $1 * 1.5\text{px}$, px会根据我们指定的不同values文件夹自动适配为合适的小。



缺点

第一，Android不同分辨率的手机实在太多了，可能你说主流就可以，的确小公司主流就可以，淘宝这种App肯定不能只适配主流手机。

第二，控件在设计图上显示的大小以及控件之间的间隙在小分辨率和大分辨率手机上天壤之别，你会发现大屏幕手机上控件超级大。可能你会觉得正常，毕竟分辨率不同。但实际效果大的有些夸张。

第三，设计图（比如360x40）上的内容占据屏幕的2/3，按照这种适配，特长手机(2960x1440 S8)上的内容也会占据2/3,这肯定不合理，控件之间

的间隙会特别大，一看就不符合设计效果，手机长，内容占据低于2/3才正常，比如可能占据1/3.第四，占据资源大：好几百KB，甚至多达1M或跟多。

3、dimen 基于dp的适配 (smallestWidth适配)

原理

这种适配依据的是最小宽度限定符。指的是Android会识别屏幕可用高度和宽度的最小尺寸的dp值（其实就是手机的宽度值），然后根据识别到的结果去资源文件中寻找对应限定符的文件夹下的资源文件。这种机制和上文提到的宽高限定符适配原理上是一样的，都是系统通过特定的规则来选择对应的文件。

举个例子，小米5的dpi是480,横向像素是1080px，根据 $px=dp(dpi/160)$ ，横向的dp值是 $1080/(480/160)$,也就是360dp,系统就会去寻找是否存在value-sw360dp的文件夹以及对应的资源文件。



```
<dimen name="base_dp">360dp</dimen>
<dimen name="qb_px_0">0.00dp</dimen>
<dimen name="qb_px_1">0.96dp</dimen>
<dimen name="qb_px_2">1.92dp</dimen>
<dimen name="qb_px_3">2.88dp</dimen>
<dimen name="qb_px_4">3.84dp</dimen>
```

smallestWidth限定符适配和宽高限定符适配最大的区别在于，有很好的容错机制，如果没有value-sw360dp文件夹，系统会向下寻找，比如离360dp最近的只有value-sw350dp，那么Android就会选择value-sw350dp文件夹下面的资源文件。这个特性就完美的解决了上文提到的宽高限定符的容错问题。

缺点

- Android 私人订制的原因，宽度方面参差不齐，不可能适配所有的手机。
- sp和dp值有些值不全（这个应该是可以解决的），姑且算是一个小问题。
- 项目中增加了N个文件夹，上拉下拉查看文件非常不方便：想看string 或者color资源文件需要拉很多再能到达。
- 通过宽度限定符就近查找的原理，可以看出来匹配出来的大小不够准确。
- 是在Android 3.2 以后引入的，Google的本意是用它来适配平板的布局文件（但是实际上显然用于diemns适配的效果更好），不过目前所有的项目应该最低支持版本应该都是4.0了（糗事百科这么老的项目最低都是4.0哦），所以，这问题其实也不重要了。

4、今日头条适配（修改手机的设备密度 density）

这套方案对老项目是不太友好的，因为修改了系统的density值之后，整个布局的实际尺寸都会发生改变，如果想要在老项目文件中使用，恐怕整个布局文件中的尺寸都可能要重新按照设计稿修改一遍才行。因此，如果你是在维护或者改造老项目，使用这套方案就要三思了。

原理

通过修改density值，强行把所有不同尺寸分辨率的手机的宽度dp值改成一个统一的值，这样就解决了所有的适配问题。

比如，设计稿宽度是360px，那么开发这边就会把目标dp值设为360dp，在不同的设备中，动态修改density值，从而保证(手机像素宽度)px/density这个值始终是360dp,这样的话，就能保证UI在不同的设备上表现一致了。

UI设计图是按屏幕宽度为360dp来设计的，那么在上述设备上，屏幕宽度其实为 $1080/(440/160)=392.7\text{dp}$ ，也就是屏幕是比设计图要宽的。这种情况下，即使使用dp也是无法在不同设备上显示为同样效果的。同时还存在部分设备屏幕宽度不足360dp，这时就会导致按360dp宽度来开发实际显示不全。加上16:9、4:3甚至其他宽高比层出不穷，宽高比不同，显示完全一致就不可能了。

通常下，我们只需要以宽或高一个维度去适配，比如我们Feed是上下滑动的，只需要保证在所有设备中宽的维度上显示一致即可，再比如一个不支持上下滑动的页面，那么需要保证在高这个维度上都显示一致，尤其不能存在某些设备上显示不全的情况。

- 支持以宽或者高一个维度去适配，保持该维度上和设计图一致；
- 支持dp和sp单位，控制迁移成本到最小。

今日头条的适配方式，今日头条适配方案默认项目中只能以高或宽中的一个作为基准，进行适配。

$\text{px} = \text{dp} * \text{density}$ (dp 是360dp)，想要 px 正好是屏幕宽度的话，只能修改 density 。

```
/\*\
 *\ 适配：修改设备密度
 */
private static float sNoncompatDensity;
private static float sNoncompatScaledDensity;

public static void setCustomDensity(@NonNull Activity
activity, @NonNull final Application application) {
    DisplayMetrics appDisplayMetrics \=
application.getResources().getDisplayMetrics();
    if (sNoncompatDensity == 0) {
        sNoncompatDensity \=
appDisplayMetrics.density;
        sNoncompatScaledDensity \=
appDisplayMetrics.scaledDensity;
        // 防止系统切换后不起作用
        application.registerComponentCallbacks(new
ComponentCallbacks() {
            @Override
            public void
onConfigurationChanged(Configuration newConfig) {
                if (newConfig != null &&
newConfig.fontScale > 0) {
```

```
sNoncompatScaledDensity \=
application.getResources().getDisplayMetrics().scaledDensity;
}

}

@Override
public void onLowMemory() {

}

});

}

float targetDensity =
appDisplayMetrics.widthPixels / 360;
// 防止字体变小
float targetScaleDensity = targetDensity \*
(sNoncompatScaledDensity / sNoncompatDensity);
int targetDensityDpi = (int) (160 \*
targetDensity);

appDisplayMetrics.density \= targetDensity;
appDisplayMetrics.scaledDensity \=
targetScaleDensity;
appDisplayMetrics.densityDpi \= targetDensityDpi;

final DisplayMetrics activityDisplayMetrics =
activity.getResources().getDisplayMetrics();
activityDisplayMetrics.density \= targetDensity;
activityDisplayMetrics.scaledDensity \=
targetScaleDensity;
activityDisplayMetrics.densityDpi \=
targetDensityDpi;

}
```

只需要在baseActivity中添加一句话即可。适配就是这么简单。

```
DisplayUtil.setCustomDensity(this, getApplication());
```

缺点

- 只需要修改一次 density，项目中的所有地方都会自动适配，这个看似解放了双手，减少了很多操作，但是实际上反应了一个缺点，那就是只能一刀切的将整个项目进行适配，但适配范围是不可控的。这样不是很好吗？这样本来是很好的，但是应用到这个方案是就不好了，因为我上面的原理也分析了，这个方案依赖于设计图尺寸，但是项目中的系统控件、三方库控件、等非我们项目自身设计的控件，它们的设计图尺寸并不会和我们项目自身的尺寸一样。当这个适配方案不分类型，将所有控件都强行使用我们项目自身的尺寸进行适配时，这时就会出现问题，当某个系统控件或三方库控件的设计图尺寸和我们项目自身的尺寸差距非常大时，这个问题就越严重。

开源库地址

[一种极低成本的Android屏幕适配方式](#)

[AndroidAutoSize](#)

1.2 Android 目前稳定高效的UI适配方案

1.2.1 概述

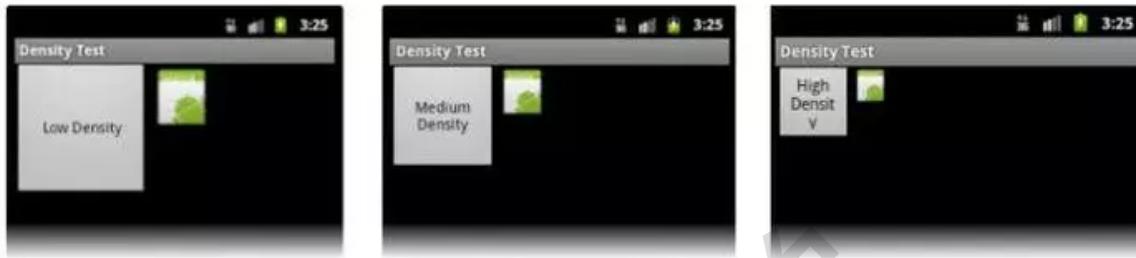
Android系统发布十多年来，关于Android的UI的适配一直是开发环节中最重要的问题，但是我看到还是有很多小伙伴对Android适配方案不了解。

刚好，近期准备对糗事百科Android客户端设计一套UI尺寸适配方案，可以和小伙伴们详细的聊一聊这个问题。

Android适配最核心的问题有两个，其一，就是适配的效率，即把设计图转化为App界面的过程是否高效，其二如何保证实现UI界面在不同尺寸和分辨率的手机中UI的一致性。

这两个问题都很重要，一个是保证我们开发的高效，一个是保证我们适配的效果；今天我们就这两个核心的问题来聊一聊Android的适配方案。

首先，大家都知道，在标识尺寸的时候，Android并不推荐我们使用px这个真实像素单位，因为不同的手机之间，分辨率是不同的，比如一个96*96像素的控件在分辨率越来越高的手机上会在整体UI中看起来越来越小。



出现类似于上图这样这样，整体的布局效果可能会变形，所以px这个单位在布局文件中是不推荐的。

1.2.2 dp 直接适配

针对这种情况，Android推荐使用dp作为尺寸单位来适配UI.

那么什么是dp？

dp指的是设备独立像素，以dp为尺寸单位的控件，在不同分辨率和尺寸的手机上代表了不同的真实像素，比如在分辨率较低的手机中，可能1dp=1px,而在分辨率较高的手机中，可能1dp=2px，这样的话，一个96*96dp的控件，在不同的手机中就能表现出差不多的大小了。

那么这个dp是如何计算的呢？

我们都应该知道一个公式： $px = dp(dpi/160)$ 系统都是通过这个来判断px和dp的数学关系，

那么这里又出现了一个问题， dpi是什么呢？

dpi是像素密度，指的是在**系统软件上**指定的单位尺寸的像素数量，它往往是写在系统出厂配置文件的一个固定值。

我为什么要强调它是软件系统上的概念？

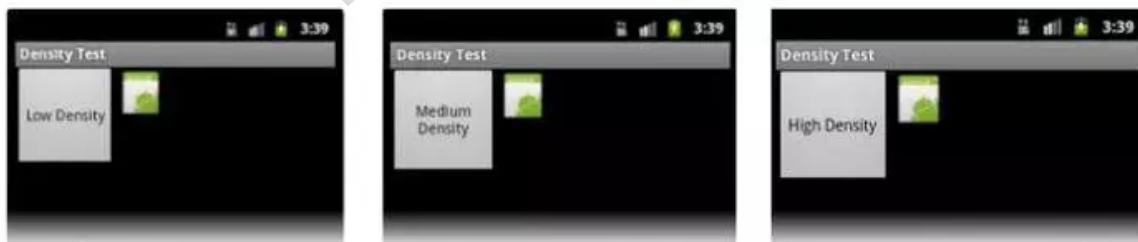
因为大家买手机的时候，往往会听到另一个叫ppi的参数，这个在手机屏幕上指的也是像素密度，但是这个是物理上的概念，它是客观存在的不会改变。dpi是软件参考了物理像素密度后，人为指定的一个值，这样保证了某一个区间内的物理像素密度在软件上都使用同一个值。这样会有利于我们的UI适配。

比如，几部相同分辨率不同尺寸的手机的ppi可能分别是430,440,450，那么在Android系统中，可能dpi会全部指定为480.这样的话， $dpi/160$ 就会是一个相对固定的数值，这样就能保证相同分辨率下不同尺寸的手机表现一致。

而在不同分辨率下，dpi将会不同，比如：

| ... | 1080*720 | 1920*1080 |
|---------|----------|-----------|
| dpi | 320 | 480 |
| dpi/160 | 2 | 3 |

根据上面的表格，我们可以发现，720P和1080P的手机，dpi是不同的，这也就意味着，不同的分辨率中，1dp对应不同数量的px(720P中， $1dp=2px$, 1080P中 $1dp=3px$)，这就实现了，当我们使用dp来定义一个控件大小的时候，他在不同的手机里表现出相应大小的像素值。

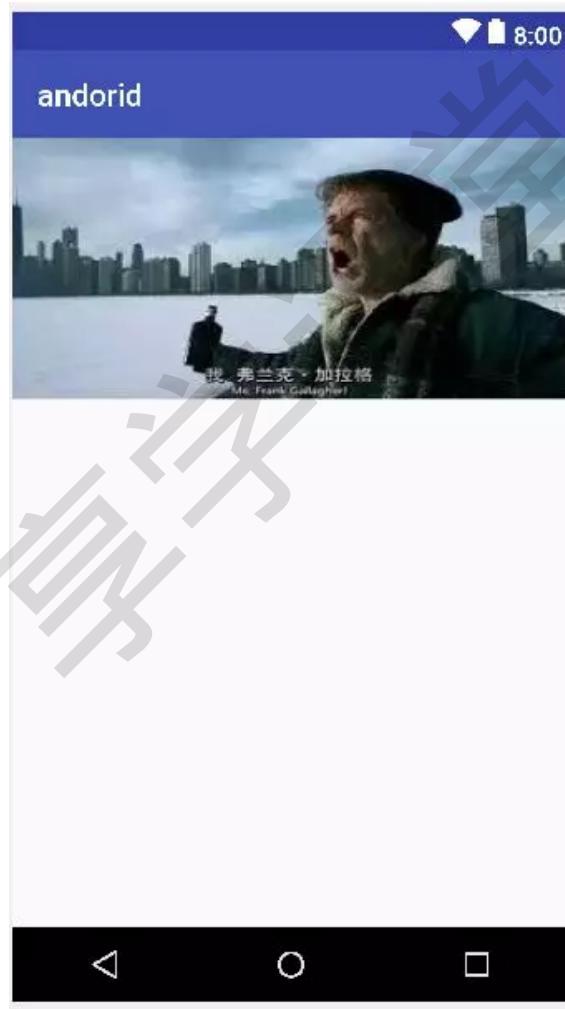


我们可以说，通过dp加上自适应布局和weight比例布局可以基本解决不同手机上适配的问题，这基本是最原始的Android适配方案。

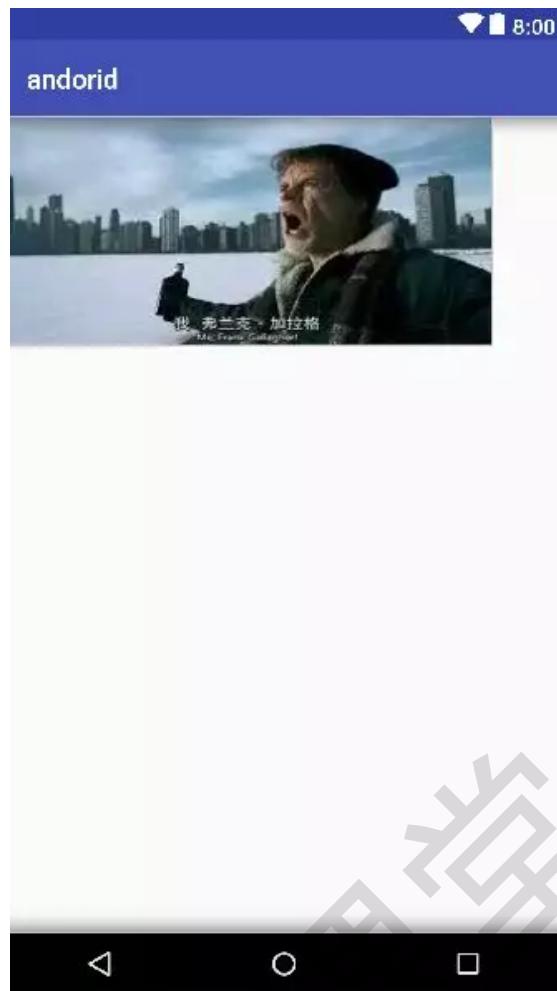
这种方式存在两个小问题，第一，这只能保证我们写出来的界面适配绝大部分手机，部分手机仍然需要单独适配，为什么dp只解决了90%的适配问题，因为并不是所有的1080P的手机dpi都是480，比如Google 的Pixel2 (1920x1080) 的dpi是420，也就是说，在Pixel2中， $1dp=2.625px$,这样会导致相同分辨率的手机中，这样，一个100dp100dp的控件，在一般的1080P手机上，可能都是300px,而Pixel 2 中，就只有262.5px,这样控件的实际大小会有所不同。

为了更形象的展示，假设我们在布局文件中把一个ImageView的宽度设置为360dp,那么在下面两张图中表现是不一样的：

图一是1080P,480dpi的手机，图二是1080P,420dpi的手机



1080P,480dpi的手机



1080P,420dpi的手机

从上面的布局中可以看到，同样是1080P的手机，差异是比较明显的。在这种情况下，我们的UI可能需要做一些微调甚至单独适配。

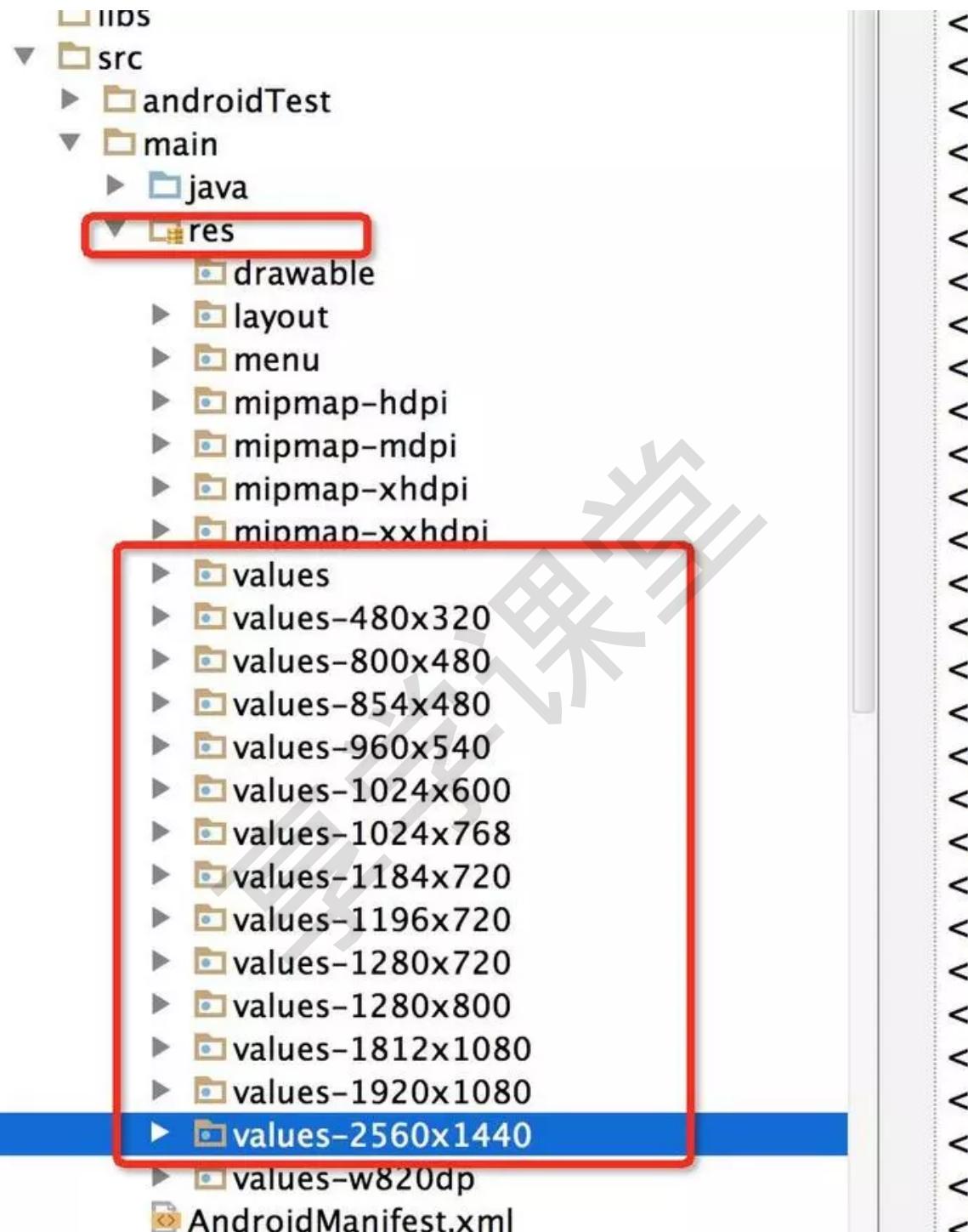
第二个问题，这种方式无法快速高效的把设计师的设计稿实现到布局代码中，通过dp直接适配，我们只能让UI基本适配不同的手机，但是在设计图和UI代码之间的鸿沟，dp是无法解决的，因为dp不是真实像素。而且，设计稿的宽高往往和Android的手机真实宽高差别极大，以我们的设计稿为例，设计稿的宽高是375px750px，而真实手机可能普遍是1080x1920，

那么在日常开发中我们是怎么跨过这个鸿沟的呢？

基本都是通过百分比啊，或者通过估算，或者设定一个规范值等等。总之，当我们拿到设计稿的时候，设计稿的ImageView是128px128px，当我们在编写layout文件的时候，却不能直接写成128dp128dp。在把设计稿向UI代码转换的过程中，我们需要耗费相当的精力去转换尺寸，这会极大的降低我们的生产力，拉低开发效率。

1.2.3 宽高限定符适配

为了高效的实现UI开发，出现了新的适配方案，我把它称作宽高限定符适配。简单说，就是穷举市面上所有的Android手机的宽高像素值：



设定一个基准的分辨率，其他分辨率都根据这个基准分辨率来计算，在不同的尺寸文件夹内部，根据该尺寸编写对应的dimens文件。

比如以480x320为基准分辨率

- 宽度为320，将任何分辨率的宽度整分为320份，取值为x1-x320
- 高度为480，将任何分辨率的高度整分为480份，取值为y1-y480

那么对于800*480的分辨率的dimens文件来说，

$$x_1 = (480/320) * 1 = 1.5\text{px}$$

$$x_2 = (480/320) * 2 = 3\text{px}$$

...

```
<?xml version="1.0" encoding="utf-8"?>
<resources><dimen name="x1">1.5px</dimen>
<dimen name="x2">3.0px</dimen>
<dimen name="x3">4.5px</dimen>
<dimen name="x4">6.0px</dimen>
<dimen name="x5">7.5px</dimen>
<dimen name="x6">9.0px</dimen>
<dimen name="x7">10.5px</dimen>
```

这个时候，如果我们的UI设计界面使用的就是基准分辨率，那么我们就可以按照设计稿上的尺寸填写相对应的dimens引用了，而当APP运行在不同分辨率的手机中时，这些系统会根据这些dimens引用去该分辨率的文件夹下面寻找对应的值。这样基本解决了我们的适配问题，而且极大的提升了我们UI开发的效率。

但是这个方案有一个致命的缺陷，那就是需要精准命中才能适配，比如1920x1080的手机就一定要找到1920x1080的限定符，否则就只能用统一的默认的dimens文件了。而使用默认的尺寸的话，UI就很可能变形，简单说，就是容错机制很差。

不过这个方案有一些团队用过，我们可以认为它是一个比较成熟有效的方案了。

1.2.4 UI适配框架（已经停止维护）

鸿洋的适配方案的项目也来自于宽高限定符方案的启发。

使用方法也很简单：

第一步：在你的项目的AndroidManifest中注明你的设计稿的尺寸。

```
<meta-data android:name="design_width"  
    android:value="768">  
</meta-data>  
<meta-data android:name="design_height"  
    android:value="1280">  
</meta-data>
```

第二步：让你的Activity继承自AutoLayoutActivity.

然后我们就可以直接在布局文件里面使用具体的像素值了，比如，设计稿上是96*96,那么我们可以直接写96px，APP运行时，框架会帮助我们根据不同手机的具体尺寸按比例伸缩。

这可以说是一个极好的方案，因为它在宽高限定舒适配的基础上更进一步，并且解决了容错机制的问题，可以说完美的达成了开发高效和适配精准的两个要求。

但是我们能够想到，因为框架要在运行时会在onMeasure里面做变换，我们自定义的控件可能会影响或限制，可能有些特定的控件，需要单独适配，这里面可能存在的暗坑是不可预见的，还有一个比较重要的问题，那就是整个适配工作是有框架完成的，而不是系统完成的，一旦使用这个框架，未来一旦遇到很难解决的问题，替换起来是非常麻烦的，而且项目一旦停止维护，后续的升级就只能靠你自己了，这种代价团队能否承受？已经停止维护了。

不过仅仅就技术方案而言，不可否认，这是一个很好的开源项目。

1.2.5今日头条适配方案（更新）

[文章链接](#)，之前确实没有接触过，我简单看了一遍，可以说，这也是相对比较完美的方案，我先简单说一下这个方案的思路，它是通过修改density值，强行把所有不同尺寸分辨率的手机的宽度dp值改成一个统一的值，这样就解决了所有的适配问题。

比如，设计稿宽度是360px，那么开发这边就会把目标dp值设为360dp，在不同的设备中，动态修改density值，从而保证(手机像素宽度)px/density这个值始终是360dp,这样的话，就能保证UI在不同的设备上表现一致了。

这个方案侵入性很低，而且也没有涉及私有API，应该也是极不错的方案，我暂时也想不到强行修改density是否会有其他影响，既然有今日头条的大厂在用，稳定性应当是有保证的。

但是根据我的观察，这套方案对老项目是不太友好的，因为修改了系统的density值之后，整个布局的实际尺寸都会发生改变，如果想要在老项目文件中使用，恐怕整个布局文件中的尺寸都可能要重新按照设计稿修改一遍才行。因此，如果你是在维护或者改造老项目，使用这套方案就要三思了。

小结

讨论的上述几种适配方案都是可以实际用于开发中的比较成熟的方案，而且确实有很多开发者正在使用。不过由于他们各自都存在一些缺陷，所以我们使用了上述方案后还需要花费额外的精力着手解决这些可能存在的缺陷。

那么，是否存在一种相对比较完美，没有明显的缺陷的方案呢？

1.2.6smallestWidth适配

smallestWidth适配，或者叫sw限定符适配。指的是Android会识别屏幕可用高度和宽度的最小尺寸的dp值（其实就是手机的宽度值），然后根据识别到的结果去资源文件中寻找对应限定符的文件夹下的资源文件。

这种机制和上文提到的宽高限定符适配原理上是一样的，都是系统通过特定的规则来选择对应的文件。

举个例子，小米5的dpi是480,横向像素是1080px，根据 $px=dp(dpi/160)$ ，横向的dp值是 $1080/(480/160)$,也就是360dp,系统就会去寻找是否存在value-sw360dp的文件夹以及对应的资源文件。

- > mipmap-xxxxapi
- > values
- > values-sw320dp
- > values-sw360dp
- > values-sw384dp
- > values-sw400dp
- > values-sw432dp
- > values-sw480dp
- > values-sw533dp
- > values-sw600dp

`smallestWidth`限定符适配和宽高限定符适配最大的区别在于，前者有很好的容错机制，如果没有`value-sw360dp`文件夹，系统会向下寻找，比如离`360dp`最近的只有`value-sw350dp`，那么Android就会选择`value-sw350dp`文件夹下面的资源文件。这个特性就完美的解决了上文提到的宽高限定符的容错问题。

这套方案是上述几种方案中最接近完美的方案。

首先，从开发效率上，它不逊色于上述任意一种方案。根据固定的放缩比例，我们基本可以按照UI设计的尺寸不假思索的填写对应的dimens引用。

我们还有以375个像素宽度的设计稿为例，在`values-sw360dp`文件夹下的dimens文件应该怎么编写呢？

这个文件夹下，意味着手机的最小宽度的dp值是360，我们把360dp等分成375等份，每一个设计稿中的像素，大概代表`smallestWidth`值为360dp的手机中的0.96dp，那么接下来的事情就很简单了，假如设计稿上出现了一个`10px*10px`的`ImageView`，那么，我们就可以不假思索的在layout文件中写下对应的尺寸。

```
<ImageView  
    android:layout_width="@dimen/qb_px_10"  
    android:layout_height="@dimen/qb_px_10" />
```

而这种dimens引用，在不同的`values-swdp`文件夹下的数值是不同的，比如`values-sw360dp`和`values-sw400dp`，

```
<dimen name="base_dpi">360dp</dimen>
<dimen name="qb_px_0">0.00dp</dimen>
<dimen name="qb_px_1">0.96dp</dimen>
<dimen name="qb_px_2">1.92dp</dimen>
<dimen name="qb_px_3">2.88dp</dimen>
<dimen name="qb_px_4">3.84dp</dimen>
<dimen name="qb_px_5">4.80dp</dimen>
<dimen name="qb_px_6">5.76dp</dimen>
<dimen name="qb_px_7">6.72dp</dimen>
<dimen name="qb_px_8">7.68dp</dimen>
<dimen name="qb_px_9">8.64dp</dimen>
<dimen name="qb_px_10">9.60dp</dimen>
<dimen name="qb_px_11">10.56dp</dimen>
<dimen name="qb_px_12">11.52dp</dimen>
<dimen name="qb_px_13">12.48dp</dimen>
<dimen name="qb_px_14">13.44dp</dimen>
<dimen name="qb_px_15">14.40dp</dimen>
<dimen name="qb_px_16">15.36dp</dimen>
<dimen name="qb_px_17">16.32dp</dimen>
<dimen name="qb_px_18">17.28dp</dimen>
<dimen name="qb_px_19">18.24dp</dimen>
<dimen name="qb_px_20">19.20dp</dimen>
...
<dimen name="base_dpi">400dp</dimen>
<dimen name="qb_px_0">0.00dp</dimen>
<dimen name="qb_px_1">1.07dp</dimen>
<dimen name="qb_px_2">2.13dp</dimen>
<dimen name="qb_px_3">3.20dp</dimen>
<dimen name="qb_px_4">4.27dp</dimen>
<dimen name="qb_px_5">5.33dp</dimen>
<dimen name="qb_px_6">6.40dp</dimen>
<dimen name="qb_px_7">7.47dp</dimen>
<dimen name="qb_px_8">8.53dp</dimen>
<dimen name="qb_px_9">9.60dp</dimen>
<dimen name="qb_px_10">10.67dp</dimen>
<dimen name="qb_px_11">11.73dp</dimen>
<dimen name="qb_px_12">12.80dp</dimen>
```

当系统识别到手机的smallestWidth值时，就会自动去寻找和目标数据最近的资源文件的尺寸。

其次，从稳定性上，它也优于上述方案。原生的dp适配可能会碰到Pixel 2这种有些特别的手机需要单独适配，但是在smallestWidth适配中，通过计算Pixel 2手机的smallestWidth的值是411，我们只需要生成一个values-sw411dp(或者取整生成values-sw410dp也没问题)就能解决问题。

smallestWidth的适配机制由系统保证，我们只需要针对这套规则生成对应的资源文件即可，不会出现什么难以解决的问题，也根本不会影响我们的业务逻辑代码，而且只要我们生成的资源文件分布合理，即使对应的smallestWidth值没有找到完全对应的资源文件，它也能向下兼容，寻找最接近的资源文件。

当然，smallestWidth适配方案有一个小问题，那就是它是在Android 3.2以后引入的，Google的本意是用它来适配平板的布局文件（但是实际上显然用于dimens适配的效果更好），不过目前所有的项目应该最低支持版本应该都是4.0了（糗事百科这么老的项目最低都是4.0哦），所以，这问题其实也不重要了。

还有一个缺陷我忘了提，那就是多个dimens文件可能导致apk变大，这是事实，根据生成的dimens文件的覆盖范围和尺寸范围，apk可能会增大300kb-800kb左右，目前糗百的dimens文件大小是406kb，我认为这是可以接受的。

项目地址：

生成dimens文件的过程以及数据计算方法上面已经讲清楚了，大家完全可以自己去生成这些文件，我在这里附赠生成values-sw的项目代码，大家直接拿去用，是Java工程。点击这里获取项目地址

https://github.com/ladingwu/dimens_sw

关于一些问题

Q: 该适配方案怎么用？

A:点击进入上文的github项目，下载到本地，然后运行该Java工程，会在本地根目录下生成相应的文件，如果需要生成更多尺寸，在DimenTypes文件中填写你需要的尺寸即可。

Q：是否有推荐的尺寸？

A 300,320,360,411, 450，这几个尺寸是比较必要的，然后在其中插入一些其他的尺寸即可，如果不放心，可以在300-450之间，以10为步长生成十几个文件。

Q:平板适配的问题？

A: 这个可以分成两个问题

第一，团队有没有专门针对平板设计UI？

第二，才是如何对平板适配。

如果团队内部没有针对平板设计UI,那么大家对于App在平板上运行的要求大抵也就是不要太难看即可。针对这种情况的适配方法是被动适配，即不要生成480以上的适配文件，这样在平板上，系统就会使用480这个尺寸的dimens文件，这样效果比主动适配更好；而如果团队主动设计了平板的UI，那么我们就需要主动生成平板的适配文件，大概在600-800之间，关键尺寸是640,768。然后按照UI设计的图来写即可。

Q：用了这套方案是否就不需要使用wrap_content等来布局了？

A:这是绝对错误的做法！如果UI设计上明显更适合使用wrap_content,match_parent,layout_weight等,我们就要毫不犹豫的使用，而且在高这个维度上，我们要依照情况设计为可滑动的方式，或者match_parent,尽量不要写死。

总之，所有的适配方案都不是用来取代match_parent,wrap_content的，而是用来完善他们的。

大家都对屏幕适配非常感兴趣，其实从我个人来说，经历过几个项目，都没有完全的引入适配方案去做这样一件事情，大部分情况下 dp就已经足够了，尤其是纵向上面，非常不希望通过屏幕高度作为基准，将满屏的内容高度限制在一个屏幕内。偶尔可能会遇到banner，一些比较特殊的地方需要用百分比，大多可以使用weight去解决，实

在不行可以硬编码一下。而且 ConstraintLayout 的出现就更加灵活了。

此外无论选择哪一种方案，都一定要提前弄清楚它的原理，因为这些方案的原理都不复杂，弄清楚有助于更合理的使用甚至扩展为更适合自己的方案。

第二节 主要控件优化

2.1 RecyclerView 优化

2.1.1 RecyclerView 性能优化

阿里四面有三面都问了这个问题，在此做了整理，希望可以帮助到大家，欢迎查漏补缺。

数据处理和视图加载分离

我们知道，从远端拉取数据肯定是要放在异步的，在我们拉取下来数据之后可能就匆匆把数据丢给了 VH 处理，其实，数据的处理逻辑我们也应该放在异步处理，这样 Adapter 在 notify change 后，ViewHolder 就可以简单无压力地做数据与视图的绑定逻辑，比如：

```
mTextView.setText(Html.fromHtml(data).toString());
```

这里的 `Html.fromHtml(data)` 方法可能就是比较耗时的，存在多个 `TextView` 的话耗时会更为严重，这样便会引发掉帧、卡顿，而如果把这一步与网络异步线程放在一起，站在用户角度，最多就是网络刷新时间稍长一点。

数据优化

分页拉取远端数据，对拉取下来的远端数据进行缓存，提升二次加载速度；对于新增或者删除数据通过 `DiffUtil` 来进行局部刷新数据，而不是一味地全局刷新数据。

布局优化

减少过渡绘制

减少布局层级，可以考虑使用自定义 View 来减少层级，或者更合理地设置布局来减少层级，不推荐在 RecyclerView 中使用 ConstraintLayout，有很多开发者已经反映了使用它效果更差，相关链接有：[Is ConstraintLayout that slow?](#)、[constraintlayout 1.1.1 not work well in listview.](#)

减少 xml 文件 inflate 时间

这里的 xml 文件不仅包括 layout 的 xml，还包括 drawable 的 xml，xml 文件 inflate 出 ItemView 是通过耗时的 IO 操作，尤其当 Item 的复用几率很低的情况下，随着 Type 的增多，这种 inflate 带来的损耗是相当大的，此时我们可以用代码去生成布局，即 new View() 的方式，只要搞清楚 xml 中每个节点的属性对应的 API 即可。

减少 View 对象的创建

一个稍微复杂的 Item 会包含大量的 View，而大量的 View 的创建也会消耗大量时间，所以要尽可能简化 itemView；设计 ItemType 时，对多 viewType 能够共用的部分尽量设计成自定义 View，减少 View 的构造和嵌套。

其他

其他并不代表不重要，而是我不能把他们进行分类哈，其中可能某些操作会对你的 RecyclerView 有很大的优化。

- 升级 Recycleview 版本到 25.1.0 及以上使用 Prefetch 功能，可参考 [RecyclerView 数据预取](#)。
- 如果 Item 高度是固定的话，可以使用 `Recyclerview.setHasFixedSize(true);` 来避免 `requestLayout` 浪费资源；
- 设置 `Recyclerview.addOnScrollListener(listener);` 来对滑动过程中停止加载的操作。

- 如果不要求动画，可以通过 `((SimpleItemAnimator) rv.getItemAnimator()).setSupportsChangeAnimations(false);` 把默认动画关闭来提神效率。
- 对 `TextView` 使用 `String.toUpperCase()` 来替代 `android:textAllCaps="true"`。
- 对 `TextView` 使用 `StaticLayout` 或者 `DynamicLayout` 的自定义 `View` 来代替它。
- 通过重写 `RecyclerView.onViewRecycled(holder)` 来回收资源。
- 通过 `RecyclerView.setItemViewCacheSize(size)` 来加大 `RecyclerView` 的缓存，用空间换时间来提高滚动的流畅性。
- 如果多个 `RecyclerView` 的 `Adapter` 是一样的，比如嵌套的 `RecyclerView` 中存在一样的 `Adapter`，可以通过设置 `RecyclerView.setRecycledViewPool(pool)` 来共用一个 `RecycledViewPool`。
- 对 `ItemView` 设置监听器，不要对每个 `Item` 都调用 `addXXListener`，应该大家公用一个 `XXListener`，根据 `ID` 来进行不同的操作，优化了对象的频繁创建带来的资源消耗。
- 通过 `getExtraLayoutSpace` 来增加 `RecyclerView` 预留的额外空间（显示范围之外，应该额外缓存的空间），如下所示：

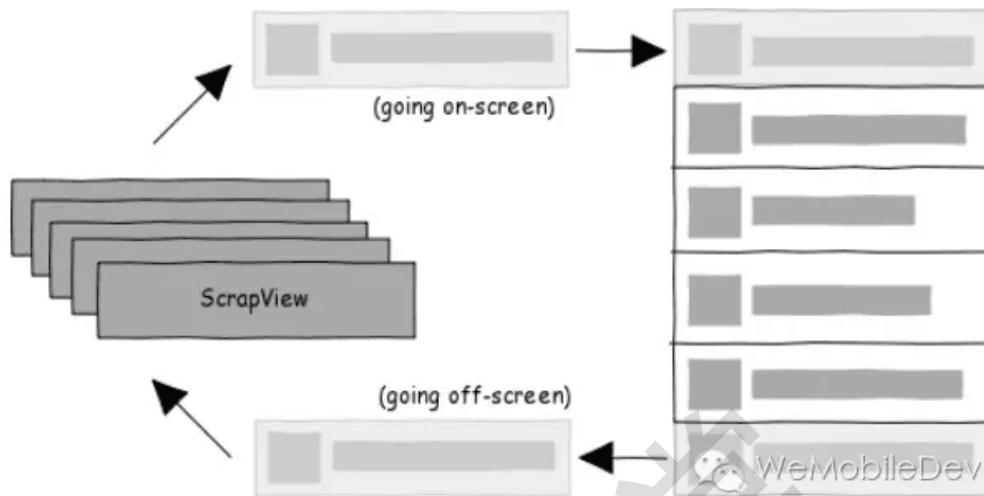
```
new LinearLayoutManager(this) {
    @Override
    protected int
    getExtraLayoutSpace(RecyclerView.State state)
    {
        return size;
    }
};
```

2.1.2 RecyclerView与ListView 对比浅析：缓存机制

2.1.2.1 背景

PS：相关知识：

ListView与RecyclerView缓存机制原理大致相似，如下图所示：



滑动过程中，离屏的ItemView即被回收至缓存，入屏的ItemView则会优先从缓存中获取，只是ListView与RecyclerView的实现细节有差异。 (这只是缓存使用的其中一个场景，还有如刷新等)

2.1.2.2 正文

2.1 缓存机制对比

1. 层级不同：

RecyclerView比ListView多两级缓存，支持多个离ItemView缓存，支持开发者自定义缓存处理逻辑，支持所有RecyclerView共用同一个RecyclerViewPool(缓存池)。

具体来说：

ListView(两级缓存)：

| ListView | | | | |
|--------------|------------------|----------------|---|-------------------|
| | 是否需要回调createView | 是否需要回调bindView | 生命周期 | 备注 |
| mActiveViews | 否 | 否 | onLayout函数周期内 | 用于屏幕内ItemView快速重用 |
| mScrapViews | 否 | 是 | 与mAdapter一致，当mAdapter被更换时，mScrapViews即被清空 | WeMobileDev |

RecyclerView(四级缓存):

| RecyclerView | | | | |
|---------------------|------------------|----------------|---|---------------------------------------|
| | 是否需要回调createView | 是否需要回调bindView | 生命周期 | 备注 |
| mAttachedScrap | 否 | 否 | onLayout函数周期内 | 用于屏幕内ItemView快速重用 |
| mCacheViews | 否 | 否 | 与mAdapter一致，当mAdapter被更换时，mCacheViews即被缓存至mRecyclerPool | 默认上限为2，即缓存屏幕外2个ItemView |
| mViewCacheExtension | | | | 不直接使用，需要用户在定制，默认不实现 |
| mRecyclerPool | 否 | 是 | 与自身生命周期一致，不再被引用时即被释放 | 默认上限为5，技术上可以实现所有RecyclerViewPool共用同一个 |

ListView和RecyclerView缓存机制基本一致：

- 1). mActiveViews和mAttachedScrap功能相似，意义在于快速重用屏幕上可见的列表项ItemView，而不需要重新createView和bindView；
- 2). mScrapView和mCachedViews + mReyclerViewPool功能相似，意义在于缓存离开屏幕的ItemView，目的是让即将进入屏幕的ItemView重用。
- 3). RecyclerView的优势在于a.mCacheViews的使用，可以做到屏幕外的

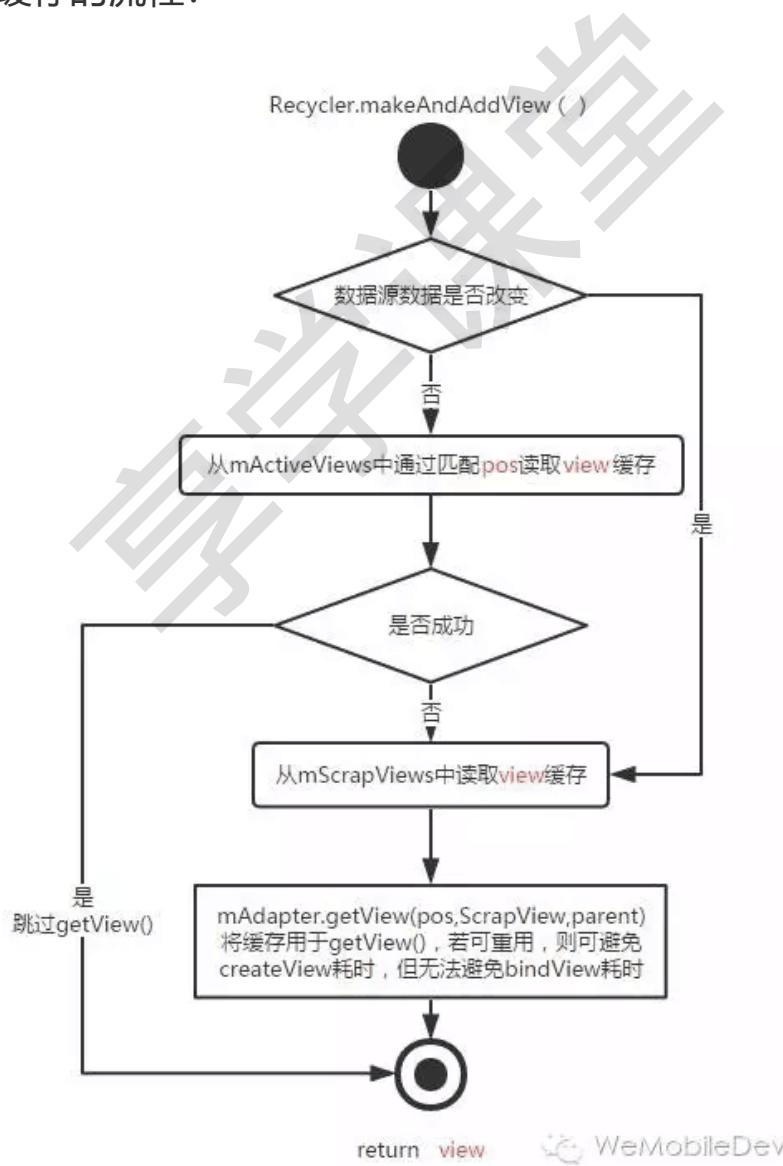
列表项ItemView进入屏幕内时也无须bindView快速重用；
b.mRecyclerPool可以供多个RecyclerView共同使用，在特定场景下，如viewpaper+多个列表页下有优势。客观来说，RecyclerView在特定场景下对ListView的缓存机制做了补强和完善。

2. 缓存不同：

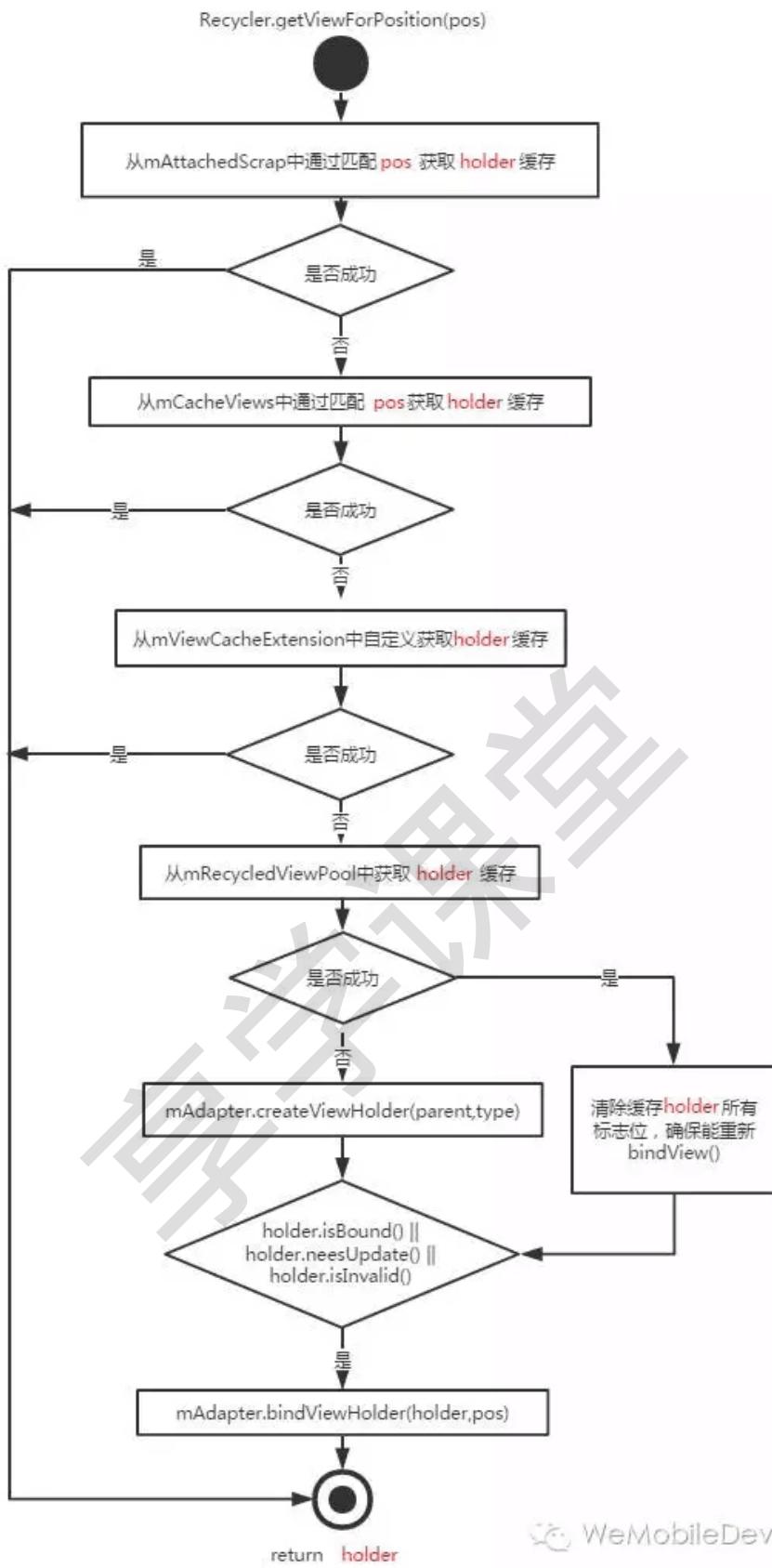
- 1). RecyclerView缓存RecyclerView.ViewHolder，抽象可理解为：
View + ViewHolder(避免每次createView时调用findViewById) + flag(标识状态);
- 2). ListView缓存View。

缓存不同，二者在缓存的使用上也略有差别，具体来说：

ListView获取缓存的流程：

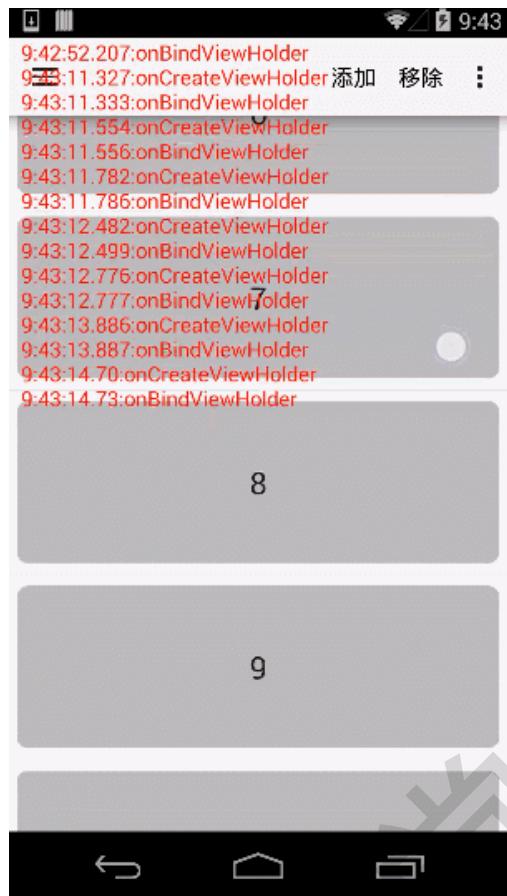


RecyclerView获取缓存的流程：



WeMobileDev

1). RecyclerView中mCacheViews(屏幕外)获取缓存时，是通过匹配pos获取目标位置的缓存，这样做好处是，当数据源数据不变的情况下，无须重新bindView：



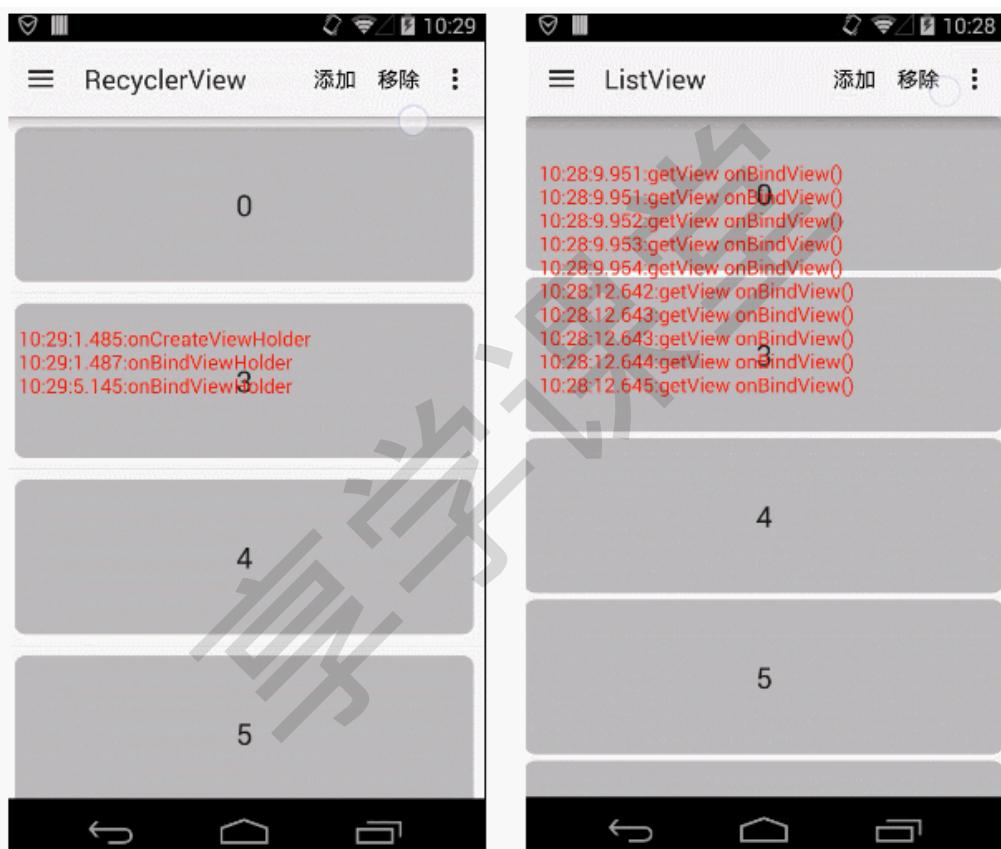
而同样是离屏缓存，ListView从mScrapViews根据pos获取相应的缓存，但是并没有直接使用，而是重新getView（即必定会重新bindView），相关代码如下：

```
//AbsListView源码: Line2345
//通过匹配pos从mScrapView中获取缓存
final View scrapview = mRecycler.getScrapview(position);
//无论是否成功都直接调用getview,导致必定会调用createview
final View child = mAdapter.getView(position, scrapview,
this);
if (scrapview != null) {
    if (child != scrapview) {
        mRecycler.addScrapview(scrapview, position);
    } else {
        ...
    }
}
```

2). ListView中通过pos获取的是view，即pos-->view；
RecyclerView中通过pos获取的是viewholder，即pos --> (view, viewHolder, flag)；
从流程图中可以看出，标志flag的作用是判断view是否需要重新 bindView，这也是RecyclerView实现局部刷新的一个核心。

2.2 局部刷新

由上文可知，RecyclerView的缓存机制确实更加完善，但还不算质的变化，RecyclerView更大的亮点在于提供了局部刷新的接口，通过局部刷新，就能避免调用许多无用的bindView。



(RecyclerView和ListView添加，移除Item效果对比)

结合RecyclerView的缓存机制，看看局部刷新是如何实现的：
以RecyclerView中notifyItemRemoved(1)为例，最终会调用
requestLayout()，使整个RecyclerView重新绘制，过程为：
onMeasure()-->onLayout()-->onDraw()

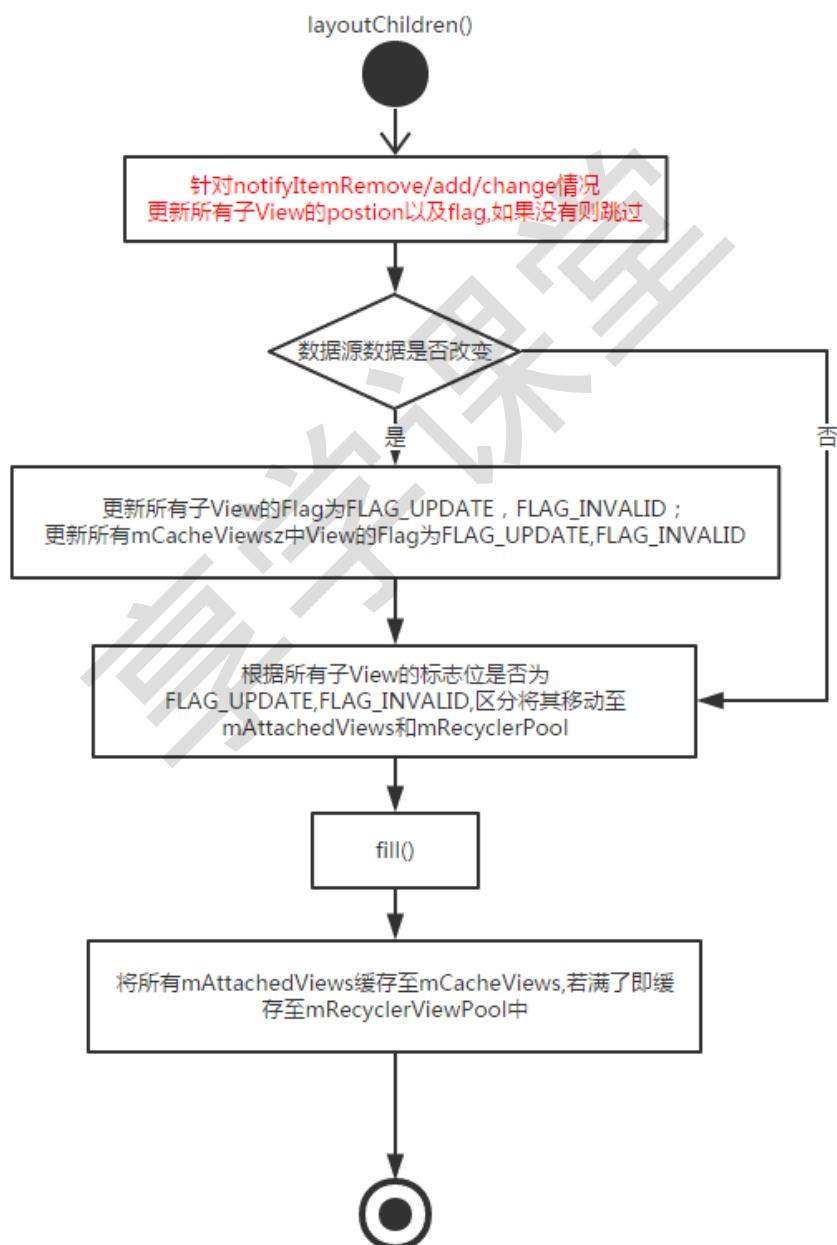
其中，onLayout()为重点，分为三步：

dispatchLayoutStep1(): 记录RecyclerView刷新前列表项ItemView的各种信息，如Top,Left,Bottom,Right，用于动画的相关计算；

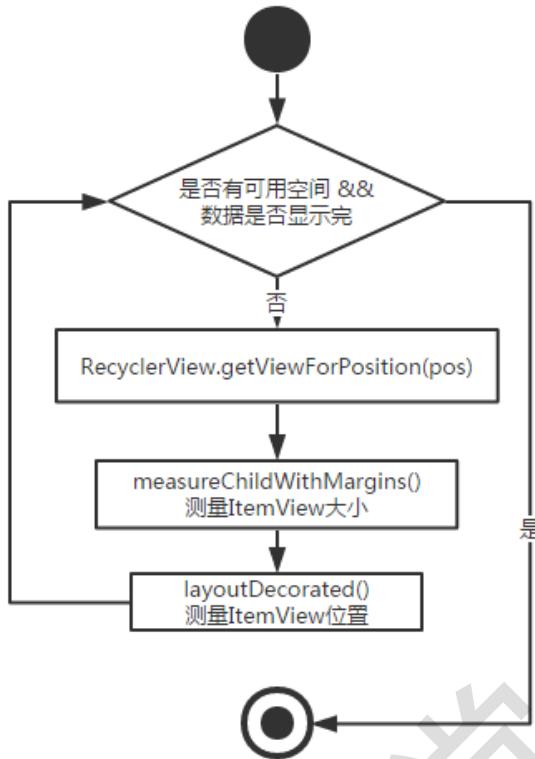
dispatchLayoutStep2(): 真正测量布局大小，位置，核心函数为layoutChildren();

dispatchLayoutStep3(): 计算布局前后各个ItemView的状态，如Remove, Add, Move, Update等，如有必要执行相应的动画.

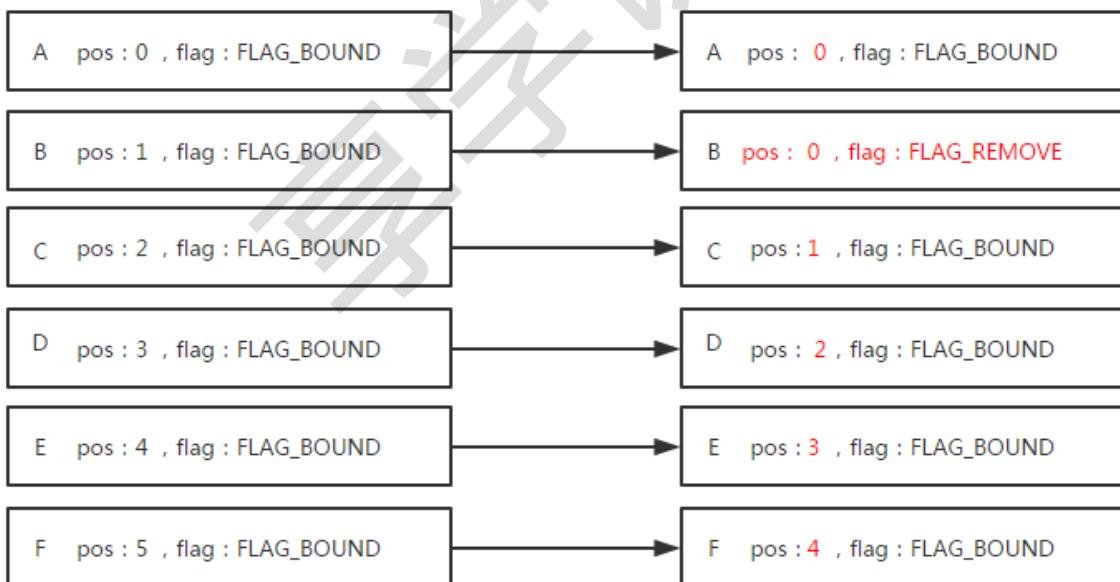
其中，layoutChildren()流程图：



RecyclerView.fill()



当调用`notifyItemRemoved`时，会对屏幕内ItemView做预处理，修改ItemView相应的pos以及flag(流程图中红色部分)：



当调用fill()中`RecyclerView.getViewForPosition(pos)`时，`RecyclerView`通过对pos和flag的预处理，使得bindview只调用一次。

需要指出，`ListView`和`RecyclerView`最大的区别在于数据源改变时的缓存的处理逻辑，`ListView`是“一锅端”，将所有的`mActiveViews`都移入了二级缓存`mScrapViews`，而`RecyclerView`则是更加灵活地对每个View修改标志位，区分是否重新`bindView`。

2.1.2.3 结论

1、在一些场景下，如界面初始化，滑动等，ListView和RecyclerView都能很好地工作，两者并没有很大的差异：

文章的开头便抛出了这样一个问题，微信Android客户端卡券模块，大部分UI都是以列表页的形式展示，实现方式为ListView，是否有必要将其替换成RecyclerView呢？



答案是否定的，从性能上看，RecyclerView并没有带来显著的提升，不需要频繁更新，暂不支持用动画，意味着RecyclerView优势也不太明显，没有太大的吸引力，ListView已经能很好地满足业务需求。

2、数据源频繁更新的场景，如弹幕：

<http://www.jianshu.com/p/2232a63442d6> 等RecyclerView的优势会非常明显；

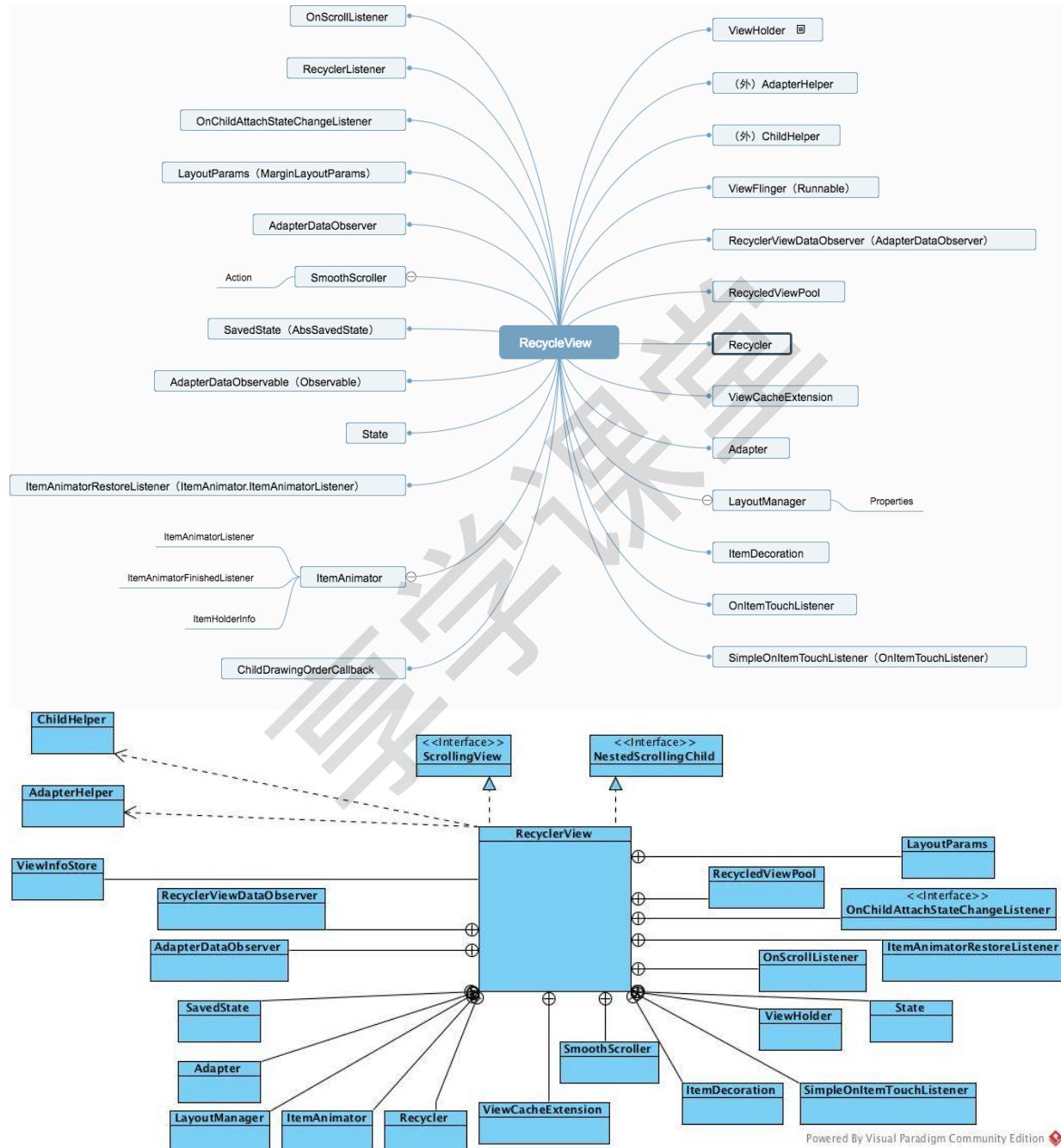
进一步来讲，结论是：

列表页展示界面，需要支持动画，或者频繁更新，局部刷新，建议使用RecyclerView，更加强大完善，易扩展；其它情况(如微信卡包列表页)两者都OK，但ListView在使用上会更加方便，快捷。

2.1.3 RecyclerView 源码分析

本文承接上文[《RecyclerView 中的设计模式》](#)，结合源码分析
Recyclerview绘制、滑动、和缓存等逻辑

RecyclerView的代码设计结构，如下面两张图：



- **RecyclerViewDataObserver** 数据观察器
- **Recycler View**循环复用系统，核心部件
- **SavedState** **RecyclerView**状态
- **AdapterHelper** 适配器更新

- ChildHelper 管理子View
- ViewInfoStore 存储子VIEW的动画信息
- Adapter 数据适配器
- LayoutManager 负责子VIEW的布局，核心部件
- ItemAnimator Item动画
- ViewFlinger 快速滑动管理
- NestedScrollingChildHelper 管理子VIEW嵌套滑动

绘制详情

可见RecyclerView涉及的类相当多，所以看代码的时候很容易迷失。因此我们需要抽丝剥茧，按照主线来进行分析。一般我们使用的时候是这样的

```
recyclerView = (RecyclerView)
        findViewById(R.id.recyclerview);
LinearLayoutManager layoutManager = new
        LinearLayoutManager(this);
//设置布局管理器
recyclerView.setLayoutManager(layoutManager);
//设置为垂直布局，这也是默认的
layoutManager.setOrientation(OrientationHelper.
        VERTICAL);
//设置Adapter
recyclerView.setAdapter(recycleAdapter);
//设置分隔线
recyclerView.addItemDecoration(new
        DividerGridItemDecoration(this));
//设置增加或删除条目的动画
recyclerView.setItemAnimator(new DefaultItemAnimator());
```

首先recyclerView = (RecyclerView) findViewById(R.id.recyclerview);会执行其构造方法,我们看一下干了些什么事:

```
public RecyclerView(Context context, @Nullable
        AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    setScrollContainer(true);
    setFocusableInTouchMode(true);
```

```
final int version = Build.VERSION.SDK_INT;
mPostUpdatesOnAnimation = version >= 16;
final ViewConfiguration vc =
ViewConfiguration.get(context);
mTouchSlop = vc.getScaledTouchSlop();
mMinFlingVelocity =
vc.getScaledMinimumFlingVelocity();
mMaxFlingVelocity =
vc.getScaledMaximumFlingVelocity();
setWillNotDraw(ViewCompat.getOverScrollMode(this) ==
ViewCompat.OVER_SCROLL_NEVER);
mItemAnimator.setListener(mItemAnimatorListener);
initAdapterManager();
initChildrenHelper();
// If not explicitly specified this view is important
for accessibility.
if (ViewCompat.getImportantForAccessibility(this)
    ==
ViewCompat.IMPORTANT_FOR_ACCESSIBILITY_AUTO) {
    ViewCompat.setImportantForAccessibility(this,
ViewCompat.IMPORTANT_FOR_ACCESSIBILITY_YES);
}
mAccessibilityManager = (AccessibilityManager)
getContext()

.getSystemService(Context.ACCESSIBILITY_SERVICE);
setAccessibilityDelegateCompat(new
RecyclerViewAccessibilityDelegate(this));
// Create the layoutManager if specified.
boolean nestedScrollingEnabled = true;
if (attrs != null) {
    int defStyleRes = 0;
    //获取布局属性值
    TypedArray a =
context.obtainStyledAttributes(attrs,
R.styleable.RecyclerView,
defStyle, defStyleRes);
```

```
        String layoutManagerName =
a.getString(R.styleable.RecyclerView_layoutManager);
        a.recycle();
        createLayoutManager(context, layoutManagerName,
attrs, defStyle, defStyleRes);
        if (Build.VERSION.SDK_INT >= 21) {
            a = context.obtainStyledAttributes(attrs,
NESTED_SCROLLING_ATTRS,
                    defStyle, defStyleRes);
            nestedScrollingEnabled = a.getBoolean(0,
true);
            a.recycle();
        }
    }
    // Re-set whether nested scrolling is enabled so that
    it is set on all API levels
    setNestedScrollingEnabled(nestedScrollingEnabled);
}
```

代码进行了一系列的初始化工作,关键是createLayoutManager,创建了一个布局管理器

```
private void createLayoutManager(Context context, String
className, AttributeSet attrs,
        int defStyleAttr, int defStyleRes) {
    //如果布局属性存在
    if (className != null) {
        className = className.trim();
        if (className.length() != 0) { // Can't use
isEmpty since it was added in API 9.
            className = getFullClassName(context,
className);
        try {
            classLoader classLoader;
            if (isInEditMode()) {
                // Stupid layoutlib cannot handle
simple class loaders.
```

```
        classLoader =
this.getClass().getClassLoader();
    } else {
        classLoader =
context.getClassLoader();
    }
    //根据布局属性值设置的layoutManager通过反射实例化layoutManager
    Class layoutManagerClass =
classLoader.loadClass(className).assubclass(LayoutManager
.class);
    Constructor constructor;
    Object[] constructorArgs = null;
    try {
        constructor = layoutManagerClass
.getConstructor(LAYOUT_MANAGER_CONSTRUCTOR_SIGNATURE);
        constructorArgs = new Object[]
{context, attrs, defStyleAttr, defStyleRes};
    } catch (NoSuchMethodException e) {
        try {
            constructor =
layoutManagerClass.getConstructor();
        } catch (NoSuchMethodException e1) {
            e1.initCause(e);
            throw new
IllegalStateException(attrs.getPositionDescription() +
": Error creating
LayoutManager " + className, e1);
        }
    }
    constructor.setAccessible(true);

setLayoutManager(constructor.newInstance(constructorArgs));
} catch (ClassNotFoundException e) {
```

```

        throw new
IllegalStateException(attrs.getPositionDescription()
                    + ": Unable to find LayoutManager
" + className, e);
    } catch (InvocationTargetException e) {
        throw new
IllegalStateException(attrs.getPositionDescription()
                    + ": Could not instantiate the
LayoutManager: " + className, e);
    } catch (InstantiationException e) {
        throw new
IllegalStateException(attrs.getPositionDescription()
                    + ": Could not instantiate the
LayoutManager: " + className, e);
    } catch (IllegalAccessException e) {
        throw new
IllegalStateException(attrs.getPositionDescription()
                    + ": Cannot access non-public
constructor " + className, e);
    } catch (ClassCastException e) {
        throw new
IllegalStateException(attrs.getPositionDescription()
                    + ": Class is not a LayoutManager
" + className, e);
    }
}
}

```

如果在布局文件里面设置了布局管理器的类型，那么这里会通过反射的方式实例化出对应的布局管理器。最后将实例化出的布局管理器设置到当前的RecyclerView,参考文章在创建实例时候

```

public void setLayoutManager(LayoutManager layout) {
    if (layout == mLayout) {
        return;
    }
    stopScroll();
}

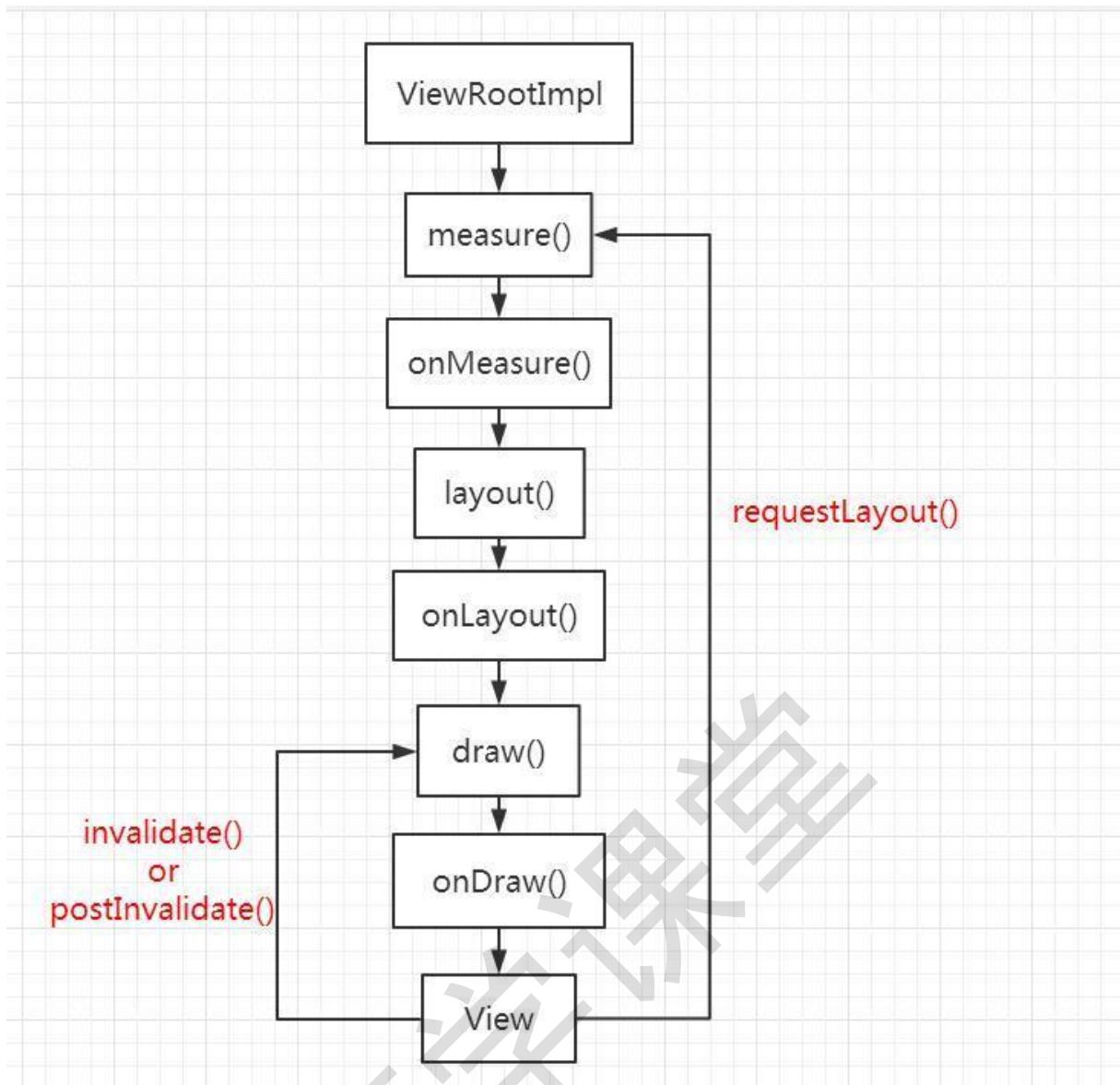
```

```
// TODO we should do this switch a dispatchLayout pass  
and animate children. There is a good  
// chance that LayoutManagers will re-use views.  
if (mLayout != null) {  
    if (mIsAttached) {  
        mLayout.dispatchDetachedFromWindow(this,  
mRecycler);  
    }  
    mLayout.setRecyclerView(null);  
}  
mRecycler.clear();  
mChildHelper.removeAllViewsUnfiltered();  
mLayout = layout;  
if (layout != null) {  
    if (layout.mRecyclerView != null) {  
        throw new  
IllegalArgumentException("LayoutManager " + layout +  
            " is already attached to a  
RecyclerView: " + layout.mRecyclerView);  
    }  
    mLayout.setRecyclerView(this);  
    if (mIsAttached) {  
        mLayout.dispatchAttachedToWindow(this);  
    }  
}  
requestLayout();  
}
```

设置布局管理器之前会先清空所有之前的缓存VIEW。最后通知VIEW刷新,requestLayout可见要绘制了

```
public void requestLayout() {  
    if (mMeasureCache != null) mMeasureCache.clear();  
    if (mAttachInfo != null &&  
mAttachInfo.mViewRequestingLayout == null) {  
        // only trigger request-during-layout logic if  
this is the view requesting it,  
        // not the views in its parent hierarchy
```

```
viewRootImpl viewRoot = getViewRootImpl();
if (viewRoot != null && viewRoot.isInLayout()) {
    if
(!viewRoot.requestLayoutDuringLayout(this)) {
        return;
    }
}
mAttachInfo.mViewRequestingLayout = this;
}
//为当前view设置标记位 PFLAG_FORCE_LAYOUT
mPrivateFlags |= PFLAG_FORCE_LAYOUT;
mPrivateFlags |= PFLAG_INVALIDATED;
if (mParent != null && !mParent.isLayoutRequested())
{
    //向父容器请求布局
    mParent.requestLayout();
}
if (mAttachInfo != null &&
mAttachInfo.mViewRequestingLayout == this) {
    mAttachInfo.mViewRequestingLayout = null;
}
}
```



在requestLayout方法中，首先先判断当前View树是否正在布局流程，接着为当前子View设置标记位，该标记位的作用就是标记了当前的View是需要进行重新布局的，接着调用mParent.requestLayout方法，这个十分重要，因为这里是向父容器请求布局，即调用父容器的requestLayout方法，为父容器添加PFLAG_FORCE_LAYOUT标记位，而父容器又会调用它的父容器的requestLayout方法，即requestLayout事件层层向上传递，直到DecorView，即根View，而根View又会传递给ViewRootImpl，也即是说子View的requestLayout事件，最终会被ViewRootImpl接收并得到处理。

纵观这个向上传递的流程，其实是采用了责任链模式，即不断向上传递该事件，直到找到能处理该事件的上级，在这里，只有ViewRootImpl能够处理requestLayout事件。

```

@Override
public void requestLayout() {

```

```
    if (!mHandlingLayoutInLayoutRequest) {
        checkThread();
        mLayoutRequested = true;
        scheduleTraversals();
    }
}

void scheduleTraversals() {
    if (!mTraversalsScheduled) {
        mTraversalsScheduled = true;
        mTraversalBarrier =
mHandler.getLooper().getQueue().postSyncBarrier();
        mChoreographer.postCallback(
            Choreographer.CallbackType.CALLBACK_TRAVERSAL,
mTraversalRunnable, null);
        if (!mUnbufferedInputDispatch) {
            scheduleConsumeBatchedInput();
        }
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}

final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        doTraversal();
    }
}

void doTraversal() {
    if (mTraversalsScheduled) {
        mTraversalsScheduled = false;

mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
        if (mProfile) {
            Debug.startMethodTracing("viewAncestor");
        }
        performTraversals();
        if (mProfile) {
```

```
        Debug.stopMethodTracing();
        mProfile = false;
    }
}
}
```

在这里，调用了scheduleTraversals方法，这个方法是一个异步方法，最终会调用到ViewRootImpl#performTraversals方法，这也是View工作流程的核心方法，在这个方法内部，分别调用measure、layout、draw方法来进行View的三大工作流程

```
private void performTraversals() {
    ...
    if (!mStopped) {
        int childwidthMeasureSpec =
getRootMeasureSpec(mWidth, lp.width); // 1
        int childHeightMeasureSpec =
getRootMeasureSpec(mHeight, lp.height);
        performMeasure(childwidthMeasureSpec,
childHeightMeasureSpec);
    }
    if (didLayout) {
        performLayout(lp, desiredWindowWidth,
desiredWindowHeight);
    ...
    }
    if (!cancelDraw && !newSurface) {
        if (!skipDraw || mReportNextDraw) {
            if (mPendingTransitions != null &&
mPendingTransitions.size() > 0) {
                for (int i = 0; i <
mPendingTransitions.size(); ++i) {

mPendingTransitions.get(i).startChangingAnimations();
            }
            mPendingTransitions.clear();
        }
    }
}
```

```
        performDraw();
    }
}

...
}
```

这里很熟悉了吧,view,viewgroup的绘制,如果这里有问题的,自己百度吧,所以会最终调用到recyclerview的onMeasure

recyclerView.setLayoutManager(layoutManager)就是桥接模式的体现,因为layoutManager的实现可以有多种,即桥接模式具体实现化逻辑ConcreteImplementor

- ListView功能 recyclerView.setLayoutManager(new LinearLayoutManager(this));
- GridView功能 recyclerView.setLayoutManager(new GridLayoutManager(this,3));
- 瀑布流形式功能 recyclerView.setLayoutManager(new StaggeredGridLayoutManager(2,StaggeredGridLayoutManager.VERTICAL));
- 横向ListView的功能 recyclerView.setLayoutManager(new LinearLayoutManager(this)); layoutManager.setOrientation(...);

```
public LinearLayoutManager(Context context, int orientation, boolean reverseLayout) {
    setOrientation(orientation);
    setReverseLayout(reverseLayout);
    setAutoMeasureEnabled(true);
}

public StaggeredGridLayoutManager(Context context, AttributeSet attrs, int defStyleAttr,
        int defStyleRes) {
    Properties properties = getProperties(context, attrs,
        defStyleAttr, defStyleRes);
    setOrientation(properties.orientation);
    setSpanCount(properties.spanCount);
    setReverseLayout(properties.reverseLayout);
```

```
    setAutoMeasureEnabled(mGapStrategy !=  
        GAP_HANDLING_NONE);  
    mLayoutState = new LayoutState();  
    createOrientationHelpers();  
}
```

GridLayoutManager继承LinearLayoutManager

可见其初始化时候会设置AutoMeasurEnabled,前面说过， RecyclerView会将测量与布局交给LayoutManager来做，并且LayoutManager有一个叫做mAutoMeasure的属性，这个属性用来控制LayoutManager是否开启自动测量，开启自动测量的话布局就交由RecyclerView使用一套默认的测量机制，否则，自定义的LayoutManager需要重写onMeasure来处理自身的测量工作。 RecyclerView目前提供的几种LayoutManager都开启了自动测量，所以这里我们关注一下自动测量部分的逻辑：

```
protected void onMeasure(int widthSpec, int heightSpec) {  
    ...  
    if (mLayout.mAutoMeasure) {  
        final int widthMode =  
            MeasureSpec.getMode(widthSpec);  
        final int heightMode =  
            MeasureSpec.getMode(heightSpec);  
        final boolean skipMeasure = widthMode ==  
            MeasureSpec.EXACTLY  
                && heightMode == MeasureSpec.EXACTLY;  
        mLayout.onMeasure(mRecycler, mState, widthSpec,  
            heightSpec);  
        if (skipMeasure || mAdapter == null) {  
            return;  
        }  
        if (mState.mLayoutStep == State.STEP_START) {  
            dispatchLayoutStep1();  
        }  
        mLayout.setMeasureSpecs(widthSpec, heightSpec);  
        mState.mIsMeasuring = true;  
        dispatchLayoutStep2();  
    }  
}
```

```
mLayout.setMeasuredDimensionFromChildren(widthSpec,  
heightSpec);  
    if (mLayout.shouldMeasureTwice()) {  
        mLayout.setMeasureSpecs(  
  
        MeasureSpec.makeMeasureSpec(getMeasuredWidth(),  
        MeasureSpec.EXACTLY),  
  
        MeasureSpec.makeMeasureSpec(getMeasuredHeight(),  
        MeasureSpec.EXACTLY));  
        mState.mIsMeasuring = true;  
        dispatchLayoutStep2();  
  
        mLayout.setMeasuredDimensionFromChildren(widthSpec,  
heightSpec);  
    }  
}  
...  
}
```

自动测量的原理如下:当RecyclerView的宽高都为EXACTLY时,可以直接设置对应的宽高,然后返回,结束测量。

补充MeasureSpec三种模式

三种模式是EXACTLY,UNSPECIFIED,AT_MOST,分别代表精确大小,不精确大小,最大值;通过MeasureSpec.getMode就可以获得该值,那么MeasureSpec到底是由什么决定的呢?MeasureSpec是由LayoutParams通过父容器的施加的规则产生的下面我们来看一看三种模式产生的情况.

- MeasureSpec.EXACTLY父容器已经精确的检测出了子View的大小,子view的大小就是MeasureSpec.getSize()的值.适用情况:
 - a.子View的LayoutParams使用具体的值(如:宽高为100dp),不管父容器的spectMode为什么,系统返回给子View的mode为EXACTLY,系统返回给子View的大小为子View额外自己指定的大小(100dp)

- b. 子View的LayoutParams采用match_parent并且父容器的mode为EXACTLY,那么子View的mode即为EXACTLY,子View大小为父容器剩余的大小
- MeasureSpec.AT_MOST父容器期望对子View的最大值做了限定适用情况:
 - c. 子View的LayoutParams采用match_parent并且父容器的mode为AT_MOST,那么子View的mode即为AT_MOST,子View大小为父容器剩余的大小
 - d. 当子View的LayoutParams采用wrap_content时并且父容器的mode为EXACTLY或者AT_MOST时,子View的Mode就为AT_MOST, 子View的specSize就为该父容器剩余的大小
- MeasureSpec.UNSPECIFIED父容器不限定大小,子View想多大就多大适应情况:
 - e. 当子View的LayoutParams采用wrap_content时并且父容器的mode为UNSPECIFIED时,子View的Mode就为UNSPECIFIED, 子View的大小不做限制

如果宽高的测量规则不是EXACTLY的,则会在onMeasure()中开始布局的处理, 这里首先要介绍一个很重要的类:

RecyclerView.State , 这个类封装了当前RecyclerView的有用信息。State的一个变量mLayoutStep表示了RecyclerView当前的布局状态, 包括STEP_START、STEP_LAYOUT、STEP_ANIMATIONS三个, 而RecyclerView的布局过程也分为三步, 其中, STEP_START表示即将开始布局, 需要调用dispatchLayoutStep1来执行第一步布局, 接下来, 布局状态变为STEP_LAYOUT, 表示接下来需要调用dispatchLayoutStep2里进行第二步布局, 同理, 第二步布局后状态变为STEP_ANIMATIONS, 需要执行第三步布局dispatchLayoutStep3。

这三个步骤的工作也各不相同, step1负责记录状态, step2负责布局, step3则与step1进行比较, 根据变化来触发动画。

RecyclerView将布局划分的如此细致必然是有其原因的, 在开启自动测量模式的情况下, RecyclerView是支持WRAP_CONTENT属性的, 比如我们可以很容易的在RecyclerView的下面放置其它的View, RecyclerView会根据子View所占大小动态调整自己的大小, 这时候, RecyclerView就会将子

控件的measure与layout提前到Recycler的onMeasure中，因为它需要确定子空间的大小与位置后，再来设置自己的大小。所以这时候就会在onMeasure中完成step1与step2。否则，就需要在onLayout中去完成整个布局过程。

综上，整个mLayout.mAutoMeasure就是在做前两步的布局，可见RecyclerView的measure与layout是紧密相关的，所以我们来赶快瞧一瞧RecyclerView是如何layout的。

我们直接看下onLayout的代码：

```
protected void onLayout(boolean changed, int l, int t,
int r, int b) {
    TraceCompat.beginSection(TRACE_ON_LAYOUT_TAG);
    dispatchLayout();
    TraceCompat.endSection();
    mFirstLayoutComplete = true;
}
```

直接追进dispatchLayout：

```
void dispatchLayout() {
    ...
    mState.mIsMeasuring = false;
    if (mState.mLayoutStep == State.STEP_START) {
        dispatchLayoutStep1();
        mLayout.setExactMeasureSpecsFrom(this);
        dispatchLayoutStep2();
    } else if (mAdapterHelper.hasUpdates() ||
    mLayout.getWidth() != getWidth() || mLayout.getHeight() !=
    getHeight()) {
        // First 2 steps are done in onMeasure but looks
        // like we have to run again due to
        // changed size.
        mLayout.setExactMeasureSpecsFrom(this);
        dispatchLayoutStep2();
    } else {
```

```
// always make sure we sync them (to ensure mode  
is exact)  
    mLayout.setExactMeasureSpecsFrom(this);  
}  
dispatchLayoutStep3();  
}
```

通过查看dispatchLayout的代码正好验证了我们前文关于RecyclerView的layout三步走原则，如果在onMeasure中已经完成了step1与step2，则只会执行step3，否则三步会依次触发。接下来一步一步的进行分析

dispatchLayoutStep1

```
private void dispatchLayoutStep1(){  
    ...  
    if (mState.mRunSimpleAnimations) {  
        int count = mChildHelper.getChildCount();  
        for (int i = 0; i < count; ++i) {  
            final ViewHolder holder =  
getViewHolderInt(mChildHelper.getChildAt(i));  
            final ItemHolderInfo animationInfo =  
mItemAnimator  
                .recordPreLayoutInformation(mState,  
holder,  
  
ItemAnimator.buildAdapterChangeFlagsForAnimations(holder)  
,  
  
holder.getUnmodifiedPayloads());  
            mViewInfoStore.addToPreLayout(holder,  
animationInfo);  
            ...  
        }  
    ...  
    mState.mLayoutStep = State.STEP_LAYOUT;  
}  
class ViewInfoStore {
```

```
private static final boolean DEBUG = false;
@VisibleForTesting
final ArrayMap mLayoutHolderMap = new ArrayMap<>();
@VisibleForTesting
final LongSparseArray mOldChangedHolders = new
LongSparseArray<>();
void clear() {
    mLayoutHolderMap.clear();
    mOldChangedHolders.clear();
}
}

public static class ItemHolderInfo {
    public int left;
    public int top;
    public int right;
    public int bottom;
    @AdapterChanges
    public int changeFlags;
    public ItemHolderInfo() {
    }
}
```

step的第一步目的就是在记录View的状态，首先遍历当前所有的View依次进行处理，mItemAnimator会根据每个View的信息封装成一个ItemHolderInfo，这个ItemHolderInfo中主要包含的就是当前View的位置状态等。然后ItemHolderInfo 就被存入mViewInfoStore中,由代码可见被存在ArrayMap和LongSparseArray中,其是对HashMap的android优化,是用两个数组来完成存储,arraymap的key可以是任意值,SparseArray的key只能为int,其核心是折半查找

注意这里调用的是mViewInfoStore的addToPreLayout方法，我们追进：

```
void addToPreLayout(viewHolder holder, ItemHolderInfo info) {
    InfoRecord record = mLayoutHolderMap.get(holder);
    if (record == null) {
        record = InfoRecord.obtain();
        mLayoutHolderMap.put(holder, record);
    }
    record.preInfo = info;
    record.flags |= FLAG_PRE;
}
```

addToPreLayout方法中会根据holder来查询InfoRecord信息，如果没有，则生成，然后将info信息赋值给InfoRecord的preInfo变量。最后标记FLAG_PRE信息，如此，完成函数。所以纵观整个layout的第一步，就是在记录当前的View信息，因为进入第二步后，View的信息就将被改变了。我们来看第二步：

dispatchLayoutStep2

```
private void dispatchLayoutStep2() {
    ...
    mLayout.onLayoutChildren(mRecycler, mState);
    ...
    mState.mLayoutStep = State.STEP_ANIMATIONS;
}
```

layout的第二步主要就是真正的去布局View了，前面也说过，RecyclerView的布局是由LayoutManager负责的，所以第二步的主要工作也都在LayoutManager中，由于每种布局的方式不一样，这里我们以常见的**LinearLayoutManager**为例。我们看其onLayoutChildren方法：

```
public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {
    ...
    if (!mAnchorInfo.mValid || mPendingScrollPosition != NO_POSITION ||
        mPendingSavedState != null) {
```

```
        updateAnchorInfoForLayout(recycler, state,
mAnchorInfo);
    }

    ...

    if (mAnchorInfo.mLayoutFromEnd) {
        firstLayoutDirection = mShouldReverseLayout ?
LayoutState.ITEM_DIRECTION_TAIL :
                LayoutState.ITEM_DIRECTION_HEAD;
    } else {
        firstLayoutDirection = mShouldReverseLayout ?
LayoutState.ITEM_DIRECTION_HEAD :
                LayoutState.ITEM_DIRECTION_TAIL;
    }

    ...

    onAnchorReady(recycler, state, mAnchorInfo,
firstLayoutDirection);

    ...

    if (mAnchorInfo.mLayoutFromEnd) {
        ...
    } else {
        // fill towards end
        updateLayoutStateToFillEnd(mAnchorInfo);
        fill(recycler, mLayoutState, state, false);
        ...

        // fill towards start
        updateLayoutStateToFillStart(mAnchorInfo);
        ...

        fill(recycler, mLayoutState, state, false);
        ...
    }
}

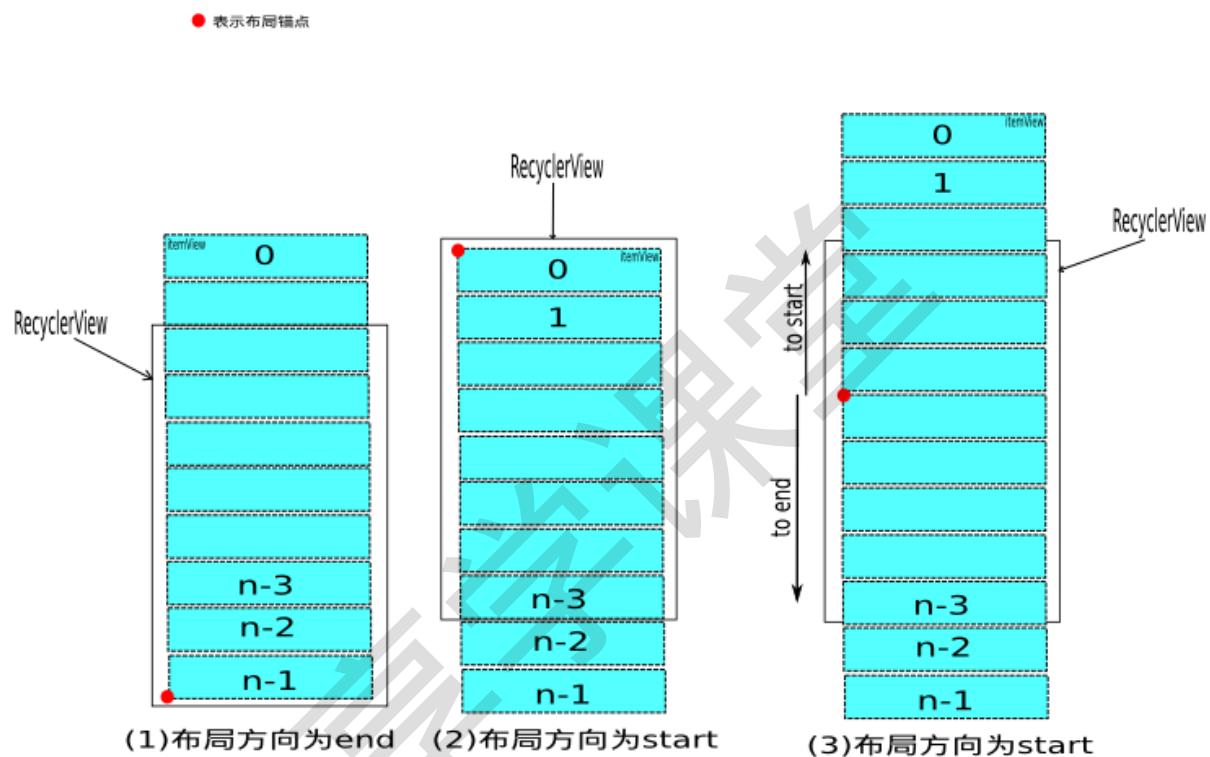
}
```

整个onLayoutChildren过程还是很复杂的，这里我尽量省略了一些与流程关系不大的细节处理代码。整个onLayoutChildren过程可以大致整理如下：

- 找到anchor点

- 根据anchor一直向前布局，直至填充满anchor点前面的所有区域
 - 根据anchor一直向后布局，直至填充满anchor点后面的所有区域
- 这里我以垂直布局来说明，`mAnchorInfo`为布局锚点信息，包含了子控件在Y轴上起始绘制偏移量（coordinate），`ItemView`在Adapter中的索引位置（position）和布局方向（`mLayoutFromEnd`）——这里是指start、end方向。

这部分代码的功能就是：确定布局锚点，以此为起点向开始和结束方向填充`ItemView`，如图所示：



anchor点的寻找是由`updateAnchorInfoForLayout`函数负责的：

```
private void
updateAnchorInfoForLayout(RecyclerView.Recycler recycler,
RecyclerView.State state,
                                         AnchorInfo
anchorInfo) {
    ...
    if (updateAnchorFromChildren(recycler, state,
anchorInfo)) {
        return;
    }
    ...
    anchorInfo.assignCoordinateFromPadding();
    anchorInfo.mPosition = mStackFromEnd ?
state.getItemCount() - 1 : 0;
}
```

函数内首先通过子view来获取anchor，如果没有获取到，就根据就取头/尾点来作为anchor。所以这里我们主要关注updateAnchorFromChildren函数：

```
private boolean
updateAnchorFromChildren(RecyclerView.Recycler recycler,
                         RecyclerView.State state, AnchorInfo anchorInfo) {
    if (getChildCount() == 0) {
        return false;
    }
    final View focused = getFocusedChild();
    if (focused != null &&
anchorInfo.isViewValidAsAnchor(focused, state)) {
        anchorInfo.assignFromViewAndKeepVisibleRect(focused);
        return true;
    }
    if (mLastStackFromEnd != mStackFromEnd) {
        return false;
    }
    View referenceChild = anchorInfo.mLayoutFromEnd
```

```
    ? findReferenceChildClosestToEnd(recycler,
state)
    : findReferenceChildClosestToStart(recycler,
state);
if (referencechild != null) {
    anchorInfo.assignFromView(referencechild);
    ...
    return true;
}
return false;
}
```

updateAnchorFromChildren内部做的事情也很容易理解，首先寻找被focus的child，找到的话以此child作为anchor，否则根据布局的方向寻找最合适的孩子来作为anchor，如果找到则将child的信息赋值给anchorInfo，其实anchorInfo主要记录的信息就是view的物理位置与在adapter中的位置。找到后返回true，否则返回false则交由上一步的函数做处理。

综上，刚刚所追踪的代码都是在寻找anchor点。在我们寻找后，LinearLayoutManager还给了我们更改anchor的时机，就是onAnchorReady函数，我们可以继承LinearLayoutManager来重写onAnchorReady方法，就可以实现某些特定的功能，比如进入RecyclerView时定位在某一项等等。

总之，我们现在找到了anchor信息，接下来就是根据anchor来布局了。无论从上到下还是从下到上布局，都调用的是fill方法，我们进入fill方法来查看一番：

```
int fill(RecyclerView.Recycler recycler, LayoutState
layoutState,
        RecyclerView.State state, boolean
stopOnFocusable) {
    final int start = layoutState.mAvailable;
    if (layoutState.mScrollingOffset !=
LayoutState.SCROLLING_OFFSET_NaN) {
        recycleByLayoutState(recycler, layoutState);
    }
}
```

```
    int remainingSpace = layoutState.mAvailable +
layoutState.mExtra;
    LayoutChunkResult layoutChunkResult =
mLayoutChunkResult;
    while ((layoutState.mInfinite || remainingSpace > 0)
&& layoutState.hasMore(state)) {
        layoutChunk(recycler, state, layoutState,
layoutChunkResult);
        ...
    }
    return start - layoutState.mAvailable;
}
```

这里同样省略了很多代码，我们关注重点：

首先比较重要的函数是recycleByLayoutState，这个函数就厉害了，它会根据当前信息对不需要的View进行回收：

```
private void recycleByLayoutState(RecyclerView.Recycler
recycler, LayoutState layoutState) {
    if (layoutState.mLayoutDirection ==
LayoutState.LAYOUT_START) {
        ...
    } else {
        recycleViewsFromStart(recycler,
layoutState.mScrollingOffset);
    }
}
```

我们继续追进recycleViewsFromStart：

```
private void recycleViewsFromStart(RecyclerView.Recycler
recycler, int dt) {
    final int limit = dt;
    final int childCount = getChildCount();
    if (mShouldReverseLayout) {
        ...
    } else {
        for (int i = 0; i < childCount; i++) {
```

```
        view child = getChildAt(i);
        if (mOrientationHelper.getDecoratedEnd(child)
> limit
        ||
mOrientationHelper.getTransformedEndwithDecoration(child)
> limit) {
            recycleChildren(recycler, 0, i);
            return;
        }
    }
}
```

这个函数的作用就是遍历所有的子View，找出逃离边界的View进行回收，回收函数我们锁定在recycleChildren里，而这个函数最后又会调到removeAndRecycleViewAt：

```
public void removeAndRecycleViewAt(int index, Recycler
recycler) {
    final View view = getChildAt(index);
    removeViewAt(index);
    recycler.recycleView(view);
}
```

这个函数首先调用removeViewAt函数，这个函数的作用是将View从RecyclerView中移除，紧接着我们看到，是recycler执行了view的回收逻辑。这里我们暂且打住，关于recycler我们会单独进行说明，这里我们只需要理解，在fill函数的一开始会去回收逃离出屏幕的view。我们再次回到fill函数，关注这里：

```
while ((layoutState.mInfinite || remainingSpace > 0) &&
layoutState.hasMore(state)) {
    layoutChunk(recycler, state, layoutState,
    layoutChunkResult);
    ...
}
```

这段代码很容易理解，只要还有剩余空间，就会执行layoutChunk方法：

```
void layoutChunk(RecyclerView.Recycler recycler,
RecyclerView.State state,
    LayoutState layoutState, LayoutChunkResult
result) {
    View view = layoutState.next(recycler);
    ...
    LayoutParams params = (LayoutParams)
view.getLayoutParams();
    if (layoutState.mScrapList == null) {
        if (mShouldReverseLayout ==
(layoutState.mLayoutDirection
        == LayoutState.LAYOUT_START)) {
            addView(view);
        } else {
            addView(view, 0);
        }
    } else {
        ...
    }
    ...
    layoutDecoratedWithMargins(view, left, top, right,
bottom);
    ...
}
```

我们首先看到，layoutState的next方法返回了一个View，凭空变出一个View，好神奇，追进去看一下：

```
View next(RecyclerView.Recycler recycler) {
    ...
    final View view =
recycler.getViewForPosition(mCurrentPosition);
    return view;
}
```

可见，view的获取逻辑也由recycler来负责，所以，这里我们同样打住，只需要清楚recycler可以根据位置返回一个view即可。

再回到layoutChunk看一下对刚刚生成的view作何处理：

```
if (mShouldReverseLayout == (layoutState.mLayoutDirection  
    == LayoutState.LAYOUT_START)) {  
    addView(view);  
} else {  
    addView(view, 0);  
}
```

很明显调用了addView方法，虽然这个方法是LayoutManager的，但这个方法最终会多次辗转调用到RecyclerView的addView方法，将view添加在RecyclerView中。综上，我们就梳理了整个第二步布局的过程，此过程完成了子View的测量与布局，任务还是相当繁重。

dispatchLayoutStep3

接下来，就到了布局的最后一步了，我们直接看下dispatchLayoutStep3方法：

```
private void dispatchLayoutStep3() {  
    mState.mLayoutStep = State.STEP_START;  
    if (mState.mRunSimpleAnimations) {  
        for (int i = mChildHelper.getChildCount() - 1; i  
        >= 0; i--) {  
            ...  
            final ItemHolderInfo animationInfo =  
                mItemAnimator  
                    .recordPostLayoutInformation(mState,  
                holder);  
            mViewInfoStore.addToPostLayout(holder,  
                animationInfo);  
        }  
        mViewInfoStore.process(mViewInfoProcessCallback);  
    }  
    ...  
}
```

这一步是与第一步呼应的，此时由于子View都已完成布局，所以子View的信息都发生了变化。我们会看到第一步出现的mViewInfoStore和mItemAnimator再次登场，这次mItemAnimator调用的是recordPostLayoutInformation方法，而mViewInfoStore调用的是addPostLayout方法，还记得刚刚我强调的吗，之前是pre，也就是真正布局之前的状态，而现在要记录布局之后的状态，我们追进addPostLayout：

```
void addPostLayout(viewHolder holder, ItemHolderInfo info) {  
    InfoRecord record = mLayoutHolderMap.get(holder);  
    if (record == null) {  
        record = InfoRecord.obtain();  
        mLayoutHolderMap.put(holder, record);  
    }  
    record.postInfo = info;  
    record.flags |= FLAG_POST;  
}
```

和第一步的addPreLayout类似，不过这次info信息被赋值给了record的postInfo变量，这样，一个record中就包含了布局前后view的状态。

最后，mViewInfoStore调用了process方法，这个方法就是根据mViewInfoStore中的View信息，来执行动画逻辑，这又是一个可以展看很多的点，这里不做探讨，感兴趣的可以深入的看一下，会对动画流程有更直观的体会。

接下来就是onDraw,RecyclerView的draw过程可以分为2部分来看：RecyclerView负责绘制所有decoration；ItemView的绘制由ViewGroup处理，这里的绘制是android常规绘制逻辑，本文就不再阐述了。下面来看看RecyclerView的draw()和onDraw()方法：

```
@Override  
public void draw(Canvas c) {  
    super.draw(c);  
    final int count = mItemDecorations.size();  
    for (int i = 0; i < count; i++) {
```

```
    mItemDecorations.get(i).onDrawOver(c, this,
mState);
}
...
}
@Override
public void onDraw(Canvas c) {
    super.onDraw(c);
    final int count = mItemDecorations.size();
    for (int i = 0; i < count; i++) {
        mItemDecorations.get(i).onDraw(c, this, mState);
    }
}
```

好了,测量,布局,绘制都大体讲了一下,回到我们开头,setLayoutManager已经完成。接下来是setAdapter(适配器模式),我们一起结合动画的实现(观察者模式)来解读,先看一下adapter类

setAdapter(适配器模式)

```
public static abstract class Adapter {
    private final AdapterDataObservable mobservable = new
AdapterDataObservable();
    public void
registerAdapterDataObserver(AdapterDataObserver observer)
{
    mobservable.registerObserver(observer);
}
    public void
unregisterAdapterDataObserver(AdapterDataObserver
observer) {
    mobservable.unregisterObserver(observer);
}
    public final void notifyItemInserted(int position) {
        mobservable.notifyItemRangeInserted(position, 1);
}
}
```

RecyclerView的Adapter，这个控件需要的是View(dst),而我们有的一般是datas(src),所以适配器Adapter就是完成了数据源datas 转化成 ItemView 的工作。带入src->Adapter->dst中，即datas->Adapter->View.

通过public abstract void onBindViewHolder(VH holder, int position);将datas绑定到view然后返回ViewHolder我们可以看到Adapter中包含一个AdapterDataObservable的对象mObservable，这个是一个可观察者，在可观察者中可以注册一系列的观察者AdapterDataObserver。在我们调用的notify函数的时候，就是可观察者发出通知，这时已经注册的观察者都可以收到这个通知，然后依次进行处理。哈哈,是不是我们前面的Subject...那么我们看一下注册观察者的地方。

注册观察者的地方就是在RecyclerView的这个函数中。这个是setAdapter方法最终调用的地方。它主要做了：

如果之前存在Adapter，先移除原来的，注销观察者，和从RecyclerView Detached。

然后根据参数，决定是否清除原来的ViewHolder

然后重置AdapterHelper，并更新Adapter，注册观察者。

```
public void setAdapter(Adapter adapter) {  
    // bail out if layout is frozen  
    setLayoutFrozen(false);  
    setAdapterInternal(adapter, false, true);  
    requestLayout();  
}
```

看一看setAdapterInternal

```
private void setAdapterInternal(Adapter adapter, boolean  
compatibleWithPrevious,  
        boolean removeAndRecycleViews) {  
    if (mAdapter != null) {  
  
        mAdapter.unregisterAdapterDataObserver(mObserver);  
        mAdapter.onDetachedFromRecyclerView(this);  
    }  
}
```

```
    if (!compatibleWithPrevious || removeAndRecycleViews)
    {
        // end all running animations
        if (mItemAnimator != null) {
            mItemAnimator.endAnimations();
        }
        // Since animations are ended, mLayout.children
        should be equal to
            // recyclerView.children. This may not be true if
        item animator's end does not work as
            // expected. (e.g. not release children
        instantly). It is safer to use mLayout's child
            // count.
        if (mLayout != null) {
            mLayout.removeAndRecycleAllViews(mRecycler);
            mLayout.removeAndRecycleScrapInt(mRecycler);
        }
        // we should clear it here before adapters are
        swapped to ensure correct callbacks.
        mRecycler.clear();
    }
    mAdapterHelper.reset();
    final Adapter oldAdapter = mAdapter;
    mAdapter = adapter;
    if (adapter != null) {
        adapter.registerAdapterDataObserver(mObserver);
        adapter.onAttachedToRecyclerview(this);
    }
    if (mLayout != null) {
        mLayout.onAdapterChanged(oldAdapter, mAdapter);
    }
    mRecycler.onAdapterChanged(oldAdapter, mAdapter,
compatibleWithPrevious);
    mState.mStructurechanged = true;
    markKnownViewsInvalid();
}
```

从这里我们可以看出，`mObserver`这个成员变量就是注册的观察者，那么我们去看看这个成员变量的内容。

该成员变量是一个`RecyclerViewDataObserver`的实例，那么`RecyclerViewDataObserver`实现了`AdapterDataObserver`中的方法。其中`onItemRangeInserted(int positionStart, int itemCount)`就是观察者接收到有数据插入通知的方法。那么我们来分析这个方法。看注释。

```
private final RecyclerViewDataObserver mObserver = new
RecyclerViewDataObserver();
private class RecyclerViewDataObserver extends
AdapterDataObserver {
    ...
    mPostUpdatesOnAnimation = version >= 16;
    @Override
    public void onItemRangeInserted(int positionStart,
int itemCount) {
        // 1) 断言不在布局或者滚动过程中，其实这就是如果在布局或者滚动过程中，则不会执
        // 行下面的内容
        assertNotInLayoutOrScroll(null);
        // 2) 这里小型，不要小看if括号中的内容，这是关键。我们去看看这个方法的实现。
        // 见下面注释 3)，在 3) 返回true之后执行
triggerUpdateProcessor方法，
        // triggerUpdateProcessor方法分析请看注释 4)。
        if
            (mAdapterHelper.onItemRangeInserted(positionStart,
itemCount)) {
                triggerUpdateProcessor();
            }
    }
}
```

`AdapterHelper`中`onItemRangeInserted`函数即相关内容，请看注释 3)。

```
class AdapterHelper implements OpReorderer.Callback {
```

```
// 一个待处理更新操作的列表，该列表中存放所有等待处理的操作信息。  
final ArrayList mPendingUpdates = new ArrayList();  
// 3) 该方法将插入操作的信息存储到一个UpdateOp中，并添加到待  
处理更新操作列表中，  
// 如果操作列表中的值是1，就返回真表示需要处理操作，等于1的判断  
避免重复触发处理操作。  
// obtainUpdateOp内部是通过池来得到一个UpdateOp对象。那么下  
面回去看我们注释 4)。  
boolean onItemRangeInserted(int positionStart, int  
itemCount) {  
    if (itemCount < 1) {  
        return false;  
    }  
    mPendingUpdates.add(obtainUpdateOp(UpdateOp.ADD,  
positionStart, itemCount, null));  
    mExistingUpdateTypes |= UpdateOp.ADD;  
    return mPendingUpdates.size() == 1;  
}  
}  
// 4) 触发更新处理操作，分为两种情况，在 版本大于16 且 已经Attach  
并且 设置了大小固定 的情况下，  
// 进行mUpdateChildviewsRunnable中的操作。否则请求布局。  
void triggerUpdateProcessor() {  
    if (mPostUpdatesOnAnimation && mHasFixedSize &&  
mIsAttached) {  
        ViewCompat.postOnAnimation(RecyclerView.this,  
mUpdateChildviewsRunnable);  
    } else {  
        mAdapterUpdateDuringMeasure = true;  
        requestLayout();  
    }  
}  
// 5) 其中核心代码是consumePendingUpdateOperations()那么继续  
往下看。  
private final Runnable mUpdateChildviewsRunnable = new  
Runnable() {  
    public void run() {
```

```
    ...
    consumePendingUpdateOperations();
}

private void consumePendingUpdateOperations() {
    ...
    if (mAdapterHelper.hasAnyUpdateTypes(UpdateOp.UPDATE)
        && !mAdapterHelper
            .hasAnyUpdateTypes(UpdateOp.ADD |
        UpdateOp.REMOVE | UpdateOp.MOVE)) {
        // 6) 如果只有更新类型的操作(这里指内容的更新, 不影响view
        // 位置的改变)的情况下,
        // 先进行预处理, 然后在没有view更新的情况下消耗延迟的更新
        // 操作, 否则调用
        // dispatchLayout方法对RecyclerView中的view重新布局。
        那么接下来分析
        // preprocess()方法。
        mAdapterHelper.preProcess();
        if (!mLayoutRequestEaten) {
            if (hasUpdatedView()) {
                dispatchLayout();
            } else {
                mAdapterHelper.consumePostponedUpdates();
            }
        }
        resumeRequestLayout(true);
    } else if (mAdapterHelper.hasPendingUpdates()) {
        // 7) 在既有更新操作又有添加或者删除或者移动中任意一个的情
        // 况下, 调用
        // dispatchLayout方法对RecyclerView中的view重新布局
        dispatchLayout();
    }
}

// 8) 预处理做了以下几件事情, <1> 先将待处理操作重排。<2> 应用所有
// 操作 <3> 清空待处理操作列表,
// 以ADD为例分析流程。
void preprocess() {
    mOpReorderer.reorderOps(mPendingUpdates);
```

```
final int count = mPendingUpdates.size();
for (int i = 0; i < count; i++) {
    UpdateOp op = mPendingUpdates.get(i);
    switch (op.cmd) {
        case UpdateOp.ADD:
            applyAdd(op);
            break;
        case UpdateOp.REMOVE:
            applyRemove(op);
            break;
        case UpdateOp.UPDATE:
            applyUpdate(op);
            break;
        case UpdateOp.MOVE:
            applyMove(op);
            break;
    }
    if (mOnItemProcessedCallback != null) {
        mOnItemProcessedCallback.run();
    }
}
mPendingUpdates.clear();
}

// 9) 直接看postponeAndUpdateviewHolders
private void applyAdd(UpdateOp op) {
    postponeAndUpdateviewHolders(op);
}

// 10) 先将操作添加到推迟的操作列表中。然后将操作的内容交给回调处理。
private void postponeAndUpdateviewHolders(UpdateOp op) {
    mPostponedList.add(op);
    switch (op.cmd) {
        case UpdateOp.ADD:
            mCallback.offsetPositionsForAdd(op.positionStart,
                op.itemCount);
            break;
        case UpdateOp.MOVE:
```

```
mCallback.offsetPositionsForMove(op.positionStart,
op.itemCount);
        break;
    case UpdateOp.REMOVE:

mCallback.offsetPositionsForRemovingLayoutOrNewView(op.p
ositionStart,
            op.itemCount);
        break;
    case UpdateOp.UPDATE:

mCallback.markViewHoldersUpdated(op.positionStart,
op.itemCount, op.payload);
        break;
    default:
        throw new IllegalArgumentException("Unknown
update op type for " + op);
    }
}

// 11) 直接看offsetPositionRecordsForInsert
@Override
public void offsetPositionsForAdd(int positionStart, int
itemCount) {
    offsetPositionRecordsForInsert(positionStart,
itemCount);
    mItemsAddedOrRemoved = true;
}

// 12) 该方法主要是便利所有的viewHolder，然后把在插入位置之后的
viewHolder的位置
// 向后移动插入的个数，最后在对Recycler中缓存的viewHolder做同样的
操作，最后申请重新布局。
void offsetPositionRecordsForInsert(int positionStart,
int itemCount) {
    final int childCount =
mChildHelper.getUnfilteredChildCount();
    for (int i = 0; i < childCount; i++) {
```

```
    final ViewHolder holder =
getViewHolderInt(mChildHelper.getUnfilteredChildAt(i));
    if (holder != null && !holder.shouldIgnore() &&
holder.mPosition >= positionStart) {
        holder.offsetPosition(itemCount, false);
        mState.mStructureChanged = true;
    }
}

mRecycler.offsetPositionRecordsForInsert(positionStart,
itemCount);
requestLayout();
}
```

前面讲过ViewInfoStore这个类是用来追踪View所要做的动画的。其中有一个内部类InfoRecord，该类用来存储ViewHolder前后的信息，以及ViewHolder状态的flag。其中还有一个非常重要的方法，process，该方法会处理所有的mLayoutHolderMap中的值，并根据其flag和前后的信息来判断ViewHolder的动作，并将这个动作反应给ProcessCallback。分别有4种行为：消失，出现，一直存在，为使用。然后交给外面去处理。

前面说过

dispatchLayoutStep3mViewInfoStore.process(mViewInfoProcessCallback);,之后我们看一下mViewInfoStore的ProcessCallback的实现
mViewInfoProcessCallback，这里只拿processAppeared做分析：

```
private final ViewInfoStore.ProcessCallback
mViewInfoProcessCallback =
    new ViewInfoStore.ProcessCallback() {
        ...
        @Override
        public void processAppeared(ViewHolder viewHolder,
                                    ItemHolderInfo preInfo, ItemHolderInfo info)
{
    animateAppearance(viewHolder, preInfo, info);
}
        ...
};

};
```

然后看一下animateAppearance方法：

```
private void animateAppearance(@NonNull ViewHolder
itemHolder,
    @Nullable ItemHolderInfo preLayoutInfo, @NonNull
ItemHolderInfo postLayoutInfo) {
    itemHolder.setIsRecyclable(false);
    if (mItemAnimator.animateAppearance(itemHolder,
    preLayoutInfo, postLayoutInfo)) {
        postAnimationRunner();
    }
}
```

该方法中不要忽略if中的内容：

`mItemAnimator.animateAppearance(itemHolder, preLayoutInfo, postLayoutInfo)` 那么进入该方法：看注释。

```
@Override
public boolean animateAppearance(@NonNull ViewHolder viewHolder,
        @Nullable ItemHolderInfo preLayoutInfo, @NonNull ItemHolderInfo postLayoutInfo) {
    // 该方法通过前后的布局信息来判断是移动还是添加。下面我们以添加
    // 为例分析
    if (preLayoutInfo != null && (preLayoutInfo.left != postLayoutInfo.left
            || preLayoutInfo.top != postLayoutInfo.top))
    {
        return animateMove(viewHolder,
                preLayoutInfo.left, preLayoutInfo.top,
                postLayoutInfo.left, postLayoutInfo.top);
    } else {
        return animateAdd(viewHolder);
    }
}
```

真正实现是在DefaultItemAnimator中：这里做了三件事情，重置holder的动画，设置显示属性，然后添加到mPendingAdditions中，mPendingAdditions是一个存储添加ViewHolder的List，表示待处理的添加动画的ViewHolder。同样在DefaultItemAnimator总也有，移动的，移除的列表。

```
@Override
public boolean animateAdd(final ViewHolder holder) {
    resetAnimation(holder);
    ViewCompat.setAlpha(holder.itemView, 0);
    mPendingAdditions.add(holder);
    return true;
}
```

最后返回true，进入if，执行postAnimationRunner方法。

```
private void postAnimationRunner() {
    if (!mPostedAnimatorRunner && mIsAttached) {
        ViewCompat.postOnAnimation(this,
mItemAnimatorRunner);
        mPostedAnimatorRunner = true;
    }
}
```

去看mItemAnimatorRunner，其中调用的ItemAnimator的runPendingAnimations方法。

```
private Runnable mItemAnimatorRunner = new Runnable() {
    @Override
    public void run() {
        if (mItemAnimator != null) {
            mItemAnimator.runPendingAnimations();
        }
        mPostedAnimatorRunner = false;
    }
}
```

然后分析runPendingAnimations方法：该方法并不难，按照移除，移动，改变，添加，依次处理之前的待处理列表中的内容。这里还是以添加的做为例子来分析，看注释。

```
public void runPendingAnimations() {
    boolean removalsPending =
!mPendingRemovals.isEmpty();
    boolean movesPending = !mPendingMoves.isEmpty();
    boolean changesPending = !mPendingChanges.isEmpty();
    boolean additionsPending =
!mPendingAdditions.isEmpty();
    if (!removalsPending && !movesPending &&
!additionsPending && !changesPending) {
        return;
    }
    for (ViewHolder holder : mPendingRemovals) {
        ...
    }
}
```

```
    }

    mPendingRemovals.clear();
    if (movesPending) {
        ...
    }
    if (changesPending) {
        ...
    }
    if (additionsPending) {
        final ArrayList<ViewHolder> additions = new ArrayList<>();
        additions.addAll(mPendingAdditions);
        mAdditionsList.add(additions);
        mPendingAdditions.clear();
        // 重要的是这个adder。其中重要的是
        animateAddImpl(holder) 方法。那么来分析这个方法。
        Runnable adder = new Runnable() {
            public void run() {
                for (ViewHolder holder : additions) {
                    animateAddImpl(holder);
                }
                additions.clear();
                mAdditionsList.remove(additions);
            }
        };
        if (removalsPending || movesPending ||
            changesPending) {
            long removeDuration = removalsPending ?
                getRemoveDuration() : 0;
            long moveDuration = movesPending ?
                getMoveDuration() : 0;
            long changeDuration = changesPending ?
                getChangeDuration() : 0;
            long totalDelay = removeDuration +
                Math.max(moveDuration, changeDuration);
            View view = additions.get(0).itemView;
            ViewCompat.postOnAnimationDelayed(view,
                adder, totalDelay);
        } else {
```

```
        adder.run();
    }
}
}
```

这个方法其实就是在通过属性动画对ViewHolder中的View做渐变动画。

```
private void animateAddImpl(final ViewHolder holder) {
    final View view = holder.itemView;
    final ViewPropertyAnimatorCompat animation =
    viewCompat.animate(view);
    mAddAnimations.add(holder);
    animation.alpha(1).setDuration(getAddDuration()).
        setListener(new VpaListenerAdapter() {
            @Override
            public void onAnimationStart(View view) {
                dispatchAddStarting(holder);
            }
            @Override
            public void onAnimationCancel(View view)
{
    viewCompat.setAlpha(view, 1);
}
            @Override
            public void onAnimationEnd(View view) {
                animation.setListener(null);
                dispatchAddFinished(holder);
                mAddAnimations.remove(holder);
                dispatchFinishedWhenDone();
            }
        }).start();
}
```

到这里，终于是触发了我们的动画。其它的动作，流程类似，细节不同而已。那么通过流程我们可以深入理解以下2点：

如果我们的RecyclerView的高度和宽度不变，那么通过手动执行setHasFixedSize(true)，可以在一定程度上减少计算，提高性能。可以在4)步的时候绕过requestLayout，只走自身的布局流程。而requestLayout是申请父控件重新布局流程，两者的计算量是不一样的。自定义ItemAnimator的时候，如果在animateAppearance，animateDisappearance.....方法中直接运行了动画，就返回false，如果是暂存起来，就返回true，然后将真正执行动画的操作放在runPendingAnimations方法中。

滑动

RecyclerView的滑动过程可以分为2个阶段：手指在屏幕上移动，使RecyclerView滑动的过程，可以称为scroll；手指离开屏幕，RecyclerView继续滑动一段距离的过程，可以称为fling。现在先看看RecyclerView的触屏事件处理onTouchEvent()方法：

```
public boolean onTouchEvent(MotionEvent e) {  
    ...  
    if (mVelocityTracker == null) {  
        mVelocityTracker = VelocityTracker.obtain();  
    }  
    ...  
    switch (action) {  
        ...  
        case MotionEvent.ACTION_MOVE: {  
            ...  
            final int x = (int)  
(MotionEventCompat.getX(e, index) + 0.5f);  
            final int y = (int)  
(MotionEventCompat.getY(e, index) + 0.5f);  
            int dx = mLasteTouchX - x;  
            int dy = mLasteTouchY - y;  
            ...  
            if (mScrollState != SCROLL_STATE_DRAGGING) {  
                ...  
                if (canScrollVertically && Math.abs(dy) >  
                    mTouchSlop) {  
                    if (dy > 0) {  
                        ...  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        dy -= mTouchSlop;
    } else {
        dy += mTouchSlop;
    }
    startScroll = true;
}
if (startScroll) {

setScrollState(SCROLL_STATE_DRAGGING);
}

}

if (mScrollState == SCROLL_STATE_DRAGGING) {
    mLastTouchX = x - mScrollOffset[0];
    mLastTouchY = y - mScrollOffset[1];
    if (scrollByInternal(
        canScrollHorizontally ? dx : 0,
        canScrollVertically ? dy : 0,
        vtev)) {

getParent().requestDisallowInterceptTouchEvent(true);
}

}

} break;
...
case MotionEvent.ACTION_UP: {
    ...
    final float yvel = canScrollVertically ?
        -
        VelocityTrackerCompat.getVelocity(mVelocityTracker,
        mScrollPointerId) : 0;
    if (!((xvel != 0 || yvel != 0) && fling((int)
        xvel, (int) yvel))) {
        setScrollState(SCROLL_STATE_IDLE);
    }
    resetTouch();
} break;
...
}
```

```
...  
}
```

这里我以垂直方向的滑动来说明。当RecyclerView接收到ACTION_MOVE事件后，会先计算出手指移动距离(dy)，并与滑动阀值(mTouchSlop)比较，当大于此阀值时将滑动状态设置为SCROLL_STATE_DRAGGING，而后调用scrollByInternal()方法，使RecyclerView滑动，这样RecyclerView的滑动的第一阶段scroll就完成了；当接收到ACTION_UP事件时，会根据之前的滑动距离与时间计算出一个初速度yvel，这步计算是由VelocityTracker实现的，然后再以此初速度，调用方法fling()，完成RecyclerView滑动的第二阶段fling。显然滑动过程中关键的方法就2个：scrollByInternal()与fling()。接下来同样以垂直线性布局来说明。先来说明scrollByInternal()，跟踪进入后，会发现它最终会调用到LinearLayoutManager.scrollBy()方法，这个过程很简单，我就不列出源码了，但是分析到这里先暂停下，去看看fling()方法：

```
public boolean fling(int velocityX, int velocityY) {  
    ...  
    mViewFlinger.fling(velocityX, velocityY);  
    ...  
}
```

有用的就这一行，其它乱七八糟的不看也罢。mViewFlinger是一个Runnable的实现ViewFlinger的对象，就是它来控件着RecyclerView的fling过程的算法的。下面来看下类ViewFlinger的一段代码：

```
void postOnAnimation() {  
    if (mEatRunOnAnimationRequest) {  
        mReschedulePostAnimationCallback = true;  
    } else {  
        removeCallbacks(this);  
        ViewCompat.postOnAnimation(RecyclerView.this,  
this);  
    }  
}  
  
public void fling(int velocityX, int velocityY) {  
    setScrollState(SCROLL_STATE_SETTLING);
```

```
mLastFlingX = mLastFlingY = 0;
mScroller.fling(0, 0, velocityX, velocityY,
    Integer.MIN_VALUE, Integer.MAX_VALUE,
Integer.MIN_VALUE, Integer.MAX_VALUE);
postOnAnimation();
}
```

可以看到，其实RecyclerView的fling是借助Scroller实现的；然后postOnAnimation()方法的作用就是在将来的某个时刻会执行我们给定的一个Runnable对象，在这里就是这个mViewFlinger对象，这部分原理我就不再深入分析了，它已经不属于本文的范围了。并且，关于Scroller的作用及原理，本文也不会作过多解释。对于这两点各位可以自行查阅，有很多文章对于作过详细阐述的。接下来看看ViewFlinger.run()方法：

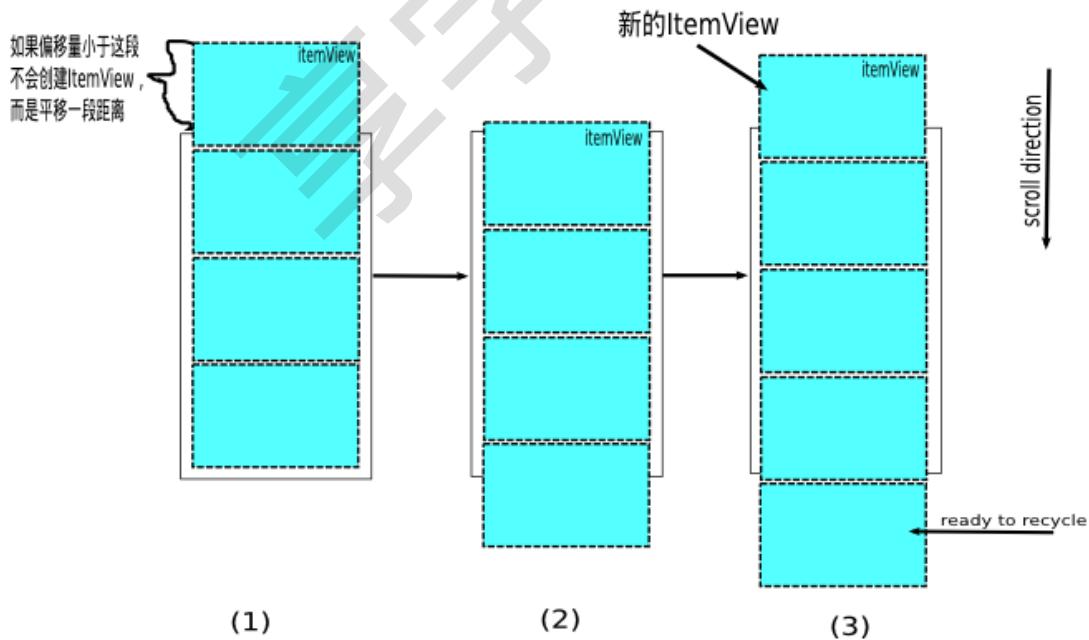
```
public void run() {
    ...
    if (scroller.computeScrollOffset()) {
        final int x = scroller.getCurrX();
        final int y = scroller.getCurrY();
        final int dx = x - mLastFlingX;
        final int dy = y - mLastFlingY;
        ...
        if (mAdapter != null) {
            ...
            if (dy != 0) {
                vresult = mLayout.scrollVerticallyBy(dy,
mRecycler, mState);
                overscrollY = dy - vresult;
            }
            ...
        }
        ...
        if (!awakenScrollBars()) {
            invalidate(); //刷新界面
        }
        ...
        if (scroller.isFinished() || !fullyConsumedAny) {
            setScrollState(SCROLL_STATE_IDLE);
        }
    }
}
```

```

        } else {
            postOnAnimation();
        }
    }
    ...
}

```

本段代码中有个方法mLayout.scrollVerticallyBy(), 跟踪进入你会发现它最终也会走到LinearLayoutManager.scrollBy(), 这样虽说RecyclerView的滑动可以分为两阶段，但是它们的实现最终其实是一样的。这里我先解释下上段代码。第一，dy表示滑动偏移量，它是由Scroller根据时间偏移量 (Scroller.fling()开始时间到当前时刻) 计算出的，当然如果是RecyclerView的scroll阶段，这个偏移量也就是手指滑动距离。第二，上段代码会多次执行，至到Scroller判断滑动结束或已经滑动到边界。再多说一下，postOnAnimation()保证了RecyclerView的滑动是流畅，这里涉及到著名的“android 16ms”机制，简单来说理想状态下，上段代码会以16毫秒一次的速度执行，这样其实，Scroller每次计算的滑动偏移量是很小的一部分，而RecyclerView就会根据这个偏移量，确定是平移ItemView，还是除了平移还需要再创建新ItemView。



现在就来看看LinearLayoutManager.scrollBy()方法：

```
int scrollBy(int dy, RecyclerView.Recycler recycler,
RecyclerView.State state) {
    ...
    final int absDy = Math.abs(dy);
    updateLayoutState(layoutDirection, absDy, true,
state);
    final int consumed = mLayoutState.mScrollingOffset
        + fill(recycler, mLayoutState, state, false);
    ...
    final int scrolled = absDy > consumed ?
layoutDirection * consumed : dy;
    mOrientationHelper.offsetChildren(-scrolled);
    ...
}
```

如上文所讲到的fill()方法，作用就是向可绘制区间填充ItemView，那么在这里，可绘制区间就是滑动偏移量！再看方法mOrientationHelper.offsetChildren()作用就是平移ItemView。好了整个滑动过程就分析完成了，当然RecyclerView的滑动还有个特性叫平滑滑动（smooth scroll），其实它的实现就是一个fling滑动，所以就不再赘述了。

缓存逻辑

前面的章节对于Recycler这个类相关的操作我们都直接进行了忽略，这里我们好好的来看下RecyclerView是如何工作的。

与ListView不同，RecyclerView的缓存是分为多级的，但其实整个的缓存逻辑还是很容易理解的，Recycler的作用就是重用ItemView。在填充ItemView的时候，ItemView是从它获取的；滑出屏幕的ItemView是由它回收的。对于不同状态的ItemView存储在了不同的集合中，比如有scrapped、cached、exCached、recycled，当然这些集合并不是都定义在同一个类里。

```
public final class Recycler {  
    // 一级缓存  
    final ArrayList<viewHolder> mAttachedScrap = new  
    ArrayList<>();  
    ArrayList<viewHolder> mChangedScrap = null;  
    // 二级缓存  
    final ArrayList<viewHolder> mCachedViews = new  
    ArrayList<viewHolder>();  
    // 三级缓存  
    private viewCacheExtension mViewCacheExtension;  
    // 四级缓存  
    RecycledviewPool mRecyclerPool;  
}
```

回到之前的layoutChunk方法中，有行代码layoutState.next(recycler)，它的作用自然就是获取itemView，我们进入这个方法查看，最终它会调用到RecyclerView.Recycler.getViewForPosition()方法：

```
view getViewForPosition(int position, boolean dryRun) {  
    return  
    tryGetViewHolderForPositionByDeadline(position, dryRun,  
    FOREVER_NS).itemView;  
}
```

getViewForPosition 可以看到其实每次 rv 取出要显示的一个item本质上就是取出一个viewholder，根据viewholder上关联的**itemview**来展示这个item。而取出viewholder最核心的方法就是
tryGetViewHolderForPositionByDeadline

```
viewHolder tryGetViewHolderForPositionByDeadline(int  
position,  
    boolean dryRun, long deadlineNs) {  
    // ...  
    boolean fromScrapOrHiddenOrCache = false;  
    viewHolder holder = null;  
    // 0) 先从 mChangedScrap 取viewholder (一级缓存)  
    if (mState.isPreLayout()) {
```

```
holder =
getChangedScrapViewForPosition(position);
fromScrapOrHiddenOrCache = holder != null;
}
// 1) 从mAttachedScrap(一级缓存) 或 mCachedViews (二级缓存) 中取viewholder
if (holder == null) {
    holder =
getScrapOrHiddenOrCachedHolderForPosition(position,
dryRun);
    if (holder != null) {
        if
(!validateViewHolderForOffsetPosition(holder)) {
            // recycle holder (and unscrap if
relevant) since it can't be used
            if (!dryRun) {
                // we would like to recycle this but
need to make sure it is not used by
                // animation logic etc.

holder.addFlags(viewHolder.FLAG_INVALID);
        if (holder.isScrap()) {

removeDetachedView(holder.itemView, false);
        holder.unScrap();
    } else if
(holder.wasReturnedFromScrap()) {

holder.clearReturnedFromScrapFlag();
    }
    recycleViewHolderInternal(holder);
}
holder = null;
} else {
    fromScrapOrHiddenOrCache = true;
}
}
}
```

```
if (holder == null) {
    // ... stable ids的逻辑
    // 3) 三级缓存
    if (holder == null && mViewCacheExtension != null) {
        // we are NOT sending the offsetPosition
        because LayoutManager does not
        // know it.
        final View view = mViewCacheExtension
            .getViewForPositionAndType(this,
position, type);
        if (view != null) {
            holder = getChildViewHolder(view);
            if (holder == null) {
                throw new
IllegalStateException("getViewForPositionAndType
returned"
                    + " a view which does not
have a viewHolder"
                    + exceptionLabel());
            } else if (holder.shouldIgnore()) {
                throw new
IllegalStateException("getViewForPositionAndType
returned"
                    + " a view that is ignored.
You must call stopIgnoring before"
                    + " returning this view." +
exceptionLabel());
            }
        }
    }
    if (holder == null) { // fallback to pool
        // 4) 四级缓存
        holder =
getRecycledViewPool().getRecycledView(type);
        if (holder != null) {
            holder.resetInternal();
            if (FORCE_INVALIDATE_DISPLAY_LIST) {
```

```
        invalidateDisplayListInt(holder);
    }
}

if (holder == null) {
    // ...
    holder =
mAdapter.createViewHolder(RecyclerView.this, type);
    // ...
}
}

//生成LayoutParams的代码 ...
return holder.itemView;
}
```

获取View的逻辑可以整理成如下：

根据列表位置获取ItemView，先后从scrapped、cached、exCached、recycled集合中查找相应的ItemView，如果没有找到，就创建（**Adapter.createViewHolder()**），最后与数据集绑定。

其中scrapped、cached和exCached集合定义在RecyclerView.Recycler中，分别表示将要在RecyclerView中删除的ItemView、一级缓存ItemView和二级缓存ItemView，cached集合的大小默认为2，exCached是需要我们通过RecyclerView.ViewCacheExtension自己实现的，默认没有；

recycled集合其实是一个Map<private SparseArray> mScrap = new SparseArray<>();，定义在RecyclerView.RecycledViewPool中，将ItemView以ItemType分类保存了下来，这里算是RecyclerView设计上的亮点，通过RecyclerView.RecycledViewPool可以实现在**不同的RecyclerView之间共享ItemView**，只要为这些不同RecyclerView设置同一个RecyclerView.RecycledViewPool就可以了。

上面解释了ItemView从不同集合中获取的方式，那么RecyclerView又是在什么时候向这些集合中添加ItemView的呢？下面我逐个介绍下。

scrapped集合中存储的其实是正在执行REMOVE操作的ItemView，这部分会在后文进一步描述。在fill()方法的循环体中有行代码

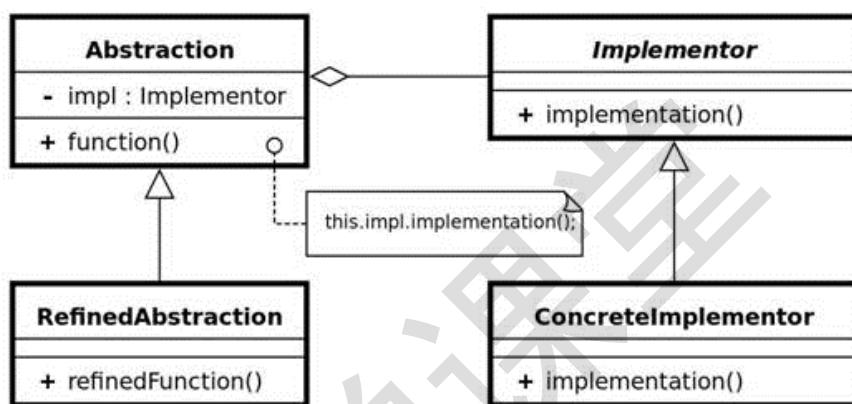
recycleByLayoutState(recycler, layoutState);，最终这个方法会执行到 RecyclerView.Recycler.recycleViewHolderInternal()方法：

```
void recycleViewHolderInternal(ViewHolder holder) {  
    ...  
    if (holder.isRecyclable()) {  
        if  
(!holder.hasAnyOfTheFlags(ViewHolder.FLAG_INVALID |  
ViewHolder.FLAG_REMOVED  
        | ViewHolder.FLAG_UPDATE)) {  
            int cachedViewSize = mCachedViews.size();  
            if (cachedViewSize >= mViewCacheMax &&  
cachedViewSize > 0) {  
                recycleCachedViewAt(0);  
                cachedViewSize --;  
            }  
            if (cachedViewSize < mViewCacheMax) {  
                mCachedViews.add(holder);  
                cached = true;  
            }  
        }  
        if (!cached) {  
            addViewHolderToRecycledViewPool(holder);  
            recycled = true;  
        }  
    }  
    ...  
}
```

View的回收并不像View的创建那么复杂，这里只涉及了两层缓存 mCachedViews与mRecyclerPool，mCachedViews相当于一个先进先出的数据结构，每当有新的View需要缓存时都会将新的View存入 mCachedViews，而mCachedViews则会移除头元素，并将头元素放入 mRecyclerPool，所以mCachedViews相当于一级缓存，mRecyclerPool 则相当于二级缓存，并且mRecyclerPool是可以多个RecyclerView共享的，这在类似于多Tab的新闻类应用会有很大的用处，因为多个Tab下的多个RecyclerView可以共用一个二级缓存。减少内存开销。

RecyclerView定义了4种针对数据集的操作，分别是ADD、 REMOVE、 UPDATE、 MOVE，封装在了AdapterHelper.UpdateOp类中，并且所有操作由一个大小为30的对象池管理着。当我们要对数据集作任何操作时，都会从这个对象池中取出一个UpdateOp对象，放入一个等待队列中，最后调用

RecyclerView.RecyclerViewDataObserver.triggerUpdateProcessor()方法，根据这个等待队列中的信息，对所有子控件重新测量、布局并绘制且执行动画。以上就是我们调用Adapter.notifyItemXXX()系列方法后发生的事。显然当我们对某个ItemView做操作时，它很有可能会影响到其它ItemView。下面我以REMOVE为例来梳理下这个流程。



首先调用Adapter.notifyItemRemove()，追溯到方法

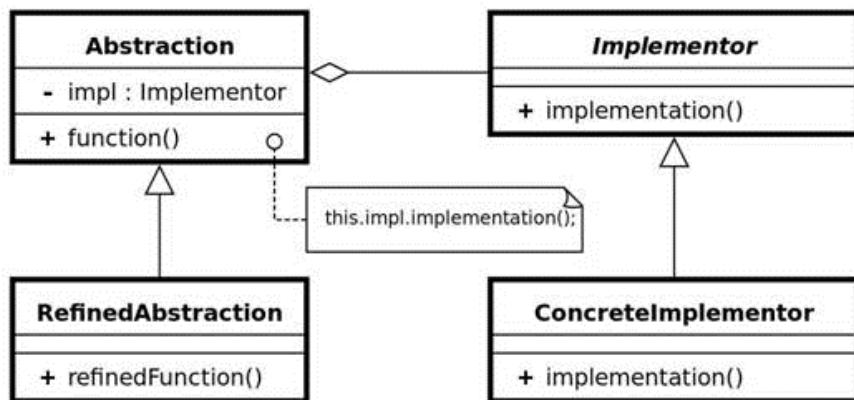
RecyclerView.RecyclerViewDataObserver.onItemRangeRemoved():

```
public void onItemRangeRemoved(int positionStart, int itemCount) {
    assertNotInLayoutOrScroll(null);
    if (mAdapterHelper.onItemRangeRemoved(positionStart, itemCount)) {
        triggerUpdateProcessor();
    }
}
```

这里的mAdapterHelper.onItemRangeRemoved()就是向之前提及的等待队列添加一个类型为REMOVE的UpdateOp对象，triggerUpdateProcessor()方法就是调用View.requestLayout()方法，这会导致界面重新布局，也就是说方法RecyclerView.onLayout()会随后调用，这之后的流程就和在绘制流程一节中所描述的一致了。

与AdapterView比较

谈到RecyclerView，总避免不了与它的前辈AdapterView家族进行一撕，这里我整理了一下RecyclerView与AdapterView的各自特点：



前面四点两位都提供了各自的实现，但也各有各自的特点：

点击事件

ListView原生提供Item单击、长按的事件，而RecyclerView则需要使用onTouchListener，相对自己实现会比较复杂。

分割线

ListView可以很轻松的设置divider属性来显示Item之间的分割线，RecyclerView需要我们自己实现ItemDecoration，前者使用简单，后者可定制性强。

布局类型

AdapterView提供了ListView与GridView两种类型，分别对应流式布局与网格式布局。RecyclerView提供了LinearLayoutManager、GridLayoutManager与之抗衡，相对而言，使用RecyclerView来进行更换布局方式更为轻松。只需要更换一个变量即可，而对于AdapterView而言则是需要更换一个View了。

缓存方式

ListView使用了一个名为RecyclerBin的类负责试图的缓存，而Recycler则使用Recycler来进行缓存，原理上两者基本一致。RecyclerView：里面存储的不是View，而是ViewHolder

局部刷新

这是一个很有用的功能，在ListView中我们想局部刷新某个Item需要自己来编写刷新逻辑，而在RecyclerView中我们可以通过
notifyItemChanged(position) 来刷新单个Item，甚至可以通过
notifyItemChanged(position, payload) 来传入一个payload信息来刷新
单个Item中的特定内容。

动画

作为视觉动物，我相信很多人喜欢RecyclerView都和它简单的动画API有关，因为之前对ListView做动画比较困难，并且不舒服。

嵌套布局

嵌套布局也是最近比较火的一个概念，RecyclerView实现了
NestedScrollingChild 接口，使得它可以和一些嵌套组件很好的工作。我们再来看ListView原生独有的几个特点：

头部与尾部的支持

ListView原生支持添加头部与尾部，虽然RecyclerView可以通过定义不同的Type来做支持，但实际应用中，如果封装的不好，是很容易出问题的，因为Adapter中的数据位置与物理数据位置发生了偏移。

多选

支持多选、单选也是ListView的一大长处，其实如果要我们自己在RecyclerView中去做支持还是需要不少代码量的。多数据源的支持
ListView提供了CursorAdapter、 ArrayAdapter，可以让我们很方便的从
数据库或者数组中获取数据，这在测试的时候很有用。

总结

综上，我们会发现RecyclerView的最大特点就是灵活，正因为这种灵活，因此会牺牲了某些便利性。而AdapterView相对来讲就比较刻板，但它原生为我们提供了很多有用的方法来便于我们快速开发。ListView并不像当年的ActivityGroup，在Fragment出来后就被标记为Deprecated。两者目前还是一种互补的关系，起码在短时间内RecyclerView还并不能完全替代AdapterView，个人感觉原因有两个，一是目前太多的应用使用了

ListView，并且ListView向RecyclerView转变也没有无损的方法。第二点，比如我就是想添加个头部，每个item带个点击事件这类简单的需求，ListView完全可以很轻松的胜任，没必要舍近求远来使用RecyclerView。因此，在实际应用中选择更适合自己的就好。

当然，从Google最近几次的更新来看，RecyclerView的进化还是很迅速的，而ListView则几乎没什么变动，所以RecyclerView绝对是大大的潜力股呀。

设计精巧的类

在翻看RecyclerView源码的过程中也遇见了许多之前没有注意过的类，这些类都可以复用在我们的日常工作当中。这里列举出其中具有代表性的几位。

Bucket

如果一个对象有大量的是与非的状态需要表示，通常我们会使用BitMask技术来节省内存，在Java中，一个byte类型，有8位(bit)，可以表达8个不同的状态，而int类型，则有32位，可以表达32种状态。再比如Long类型，有64位，则可以表达64中状态。一般情况下使用一个Long已经足够我们使用了。但如果有不设上限的状态需要我们表示呢？在ChildHelper里有一个静态内部类Bucket，基本源码如下：

```
static class Bucket {  
    final static int BITS_PER_WORD = Long.SIZE;  
    final static long LAST_BIT = 1L << (Long.SIZE - 1);  
    long mData = 0;  
    Bucket next;  
    void set(int index) {  
        if (index >= BITS_PER_WORD) {  
            ensureNext();  
            next.set(index - BITS_PER_WORD);  
        } else {  
            mData |= 1L << index;  
        }  
    }  
    ...  
}
```

可以看到，Bucket是一个链表结构，当index大于64的时候，它便会去下一个Bucket去寻找，所以，Bucket可以不设上限的表示状态。

Pools

熟悉Message回收机制的朋友可能了解，在使用Message对象时最好通过Message.obtain()方法来获取，这样可以在很多情况下避免创建新对象。在使用完之后调用 message.recycle() 来回收消息。谷歌为这种机制也提供了抽象的实现，就是位于v4包下Pools类，内部接口Pool提供了 acquire 与 release 两个方法，不过需要注意的是这个 acquire 方法可能返回空，毕竟Pools不是业务类，它不应该清楚对象的具体创建逻辑。还有一点是Pools 与 Message类的实现机制不同，每个Message对象内部都持有一个引用下一个message的指针，相当于一个链表结构，而Pool的实现类 SimplePool 中使用的是数组。Pool机制在 RecycleView 中有如下几处应用：

RecycleView将item的增删改封装为 UpdateOp 类。

ViewInfoStore 类中静态内部类 InfoRecord。

总结

RecyclerView也不是万能的，它的灵活性也是有一定限制的，比如我就遇到了一不是很好解决的问题：Recyler的缓存级别是一个Item的整个View，而我们没办法自定义缓存级别，这样说比较抽象，举个例子，我的某些Item的某个子View加载很耗时，所以我希望我在上下滑动的时候，Item的其它View是可以被回收利用的，但这个加载很耗时的View是不要重复使用的。即我希望用空间换取时间来获取滑动的流畅性。当然，这样的需求不常见，RecyclerView也不能很好的满足这一点。

RecyclerView也应该算作一个明星控件了，自从其诞生开始就备受欢迎，仔细的学习也能让我们在工作中更容易的、更恰当的使用。本文也只是分析了RecyclerView的一部分，关于动画、滑动、嵌套滑动等等还需要大家自行去研究。有兴趣的可以看看[RecyclerView 和 ListView 使用对比分析](#)

2.1.4 RecyclerView 预加载机制源码分析

RecyclerView四级缓存

再次总结一下 RecyclerView 的四级缓存

- Scrap
 - Cache
 - ViewCacheExtension
 - RecycledViewPool
- Scrap:** 对应ListView 的Active View，就是屏幕内的缓存数据，就是相当于换了个名字，可以直接拿来复用。

Cache: 刚刚移出屏幕的缓存数据，**默认大小是2个**，当其容量被充满同时又有新的数据添加的时候，会根据FIFO原则，把先进入的缓存数据移出并放到下一级缓存中，然后再把新的数据添加进来。Cache里面的数据是干净的，也就是携带了原来的ViewHolder的所有数据信息，数据可以直接拿来复用。需要注意的是，**cache是根据position来寻找数据的**，这个position是根据第一个或者最后一个可见的item的position以及用户操作行为（上拉还是下拉）。

举个栗子：当前屏幕内第一个可见的item的position是1，用户进行了一个下拉操作，那么当前预测的position就相当于（1-1=0），也就是position=0的那个item要被拉回到屏幕，此时RecyclerView就从Cache里面找position=0的数据，如果找到了就直接拿来复用。

ViewCacheExtension: 是google留给开发者自己来自定义缓存的，这个ViewCacheExtension我个人建议还是要慎用，因为我扒拉扒拉网上其他的博客，没有找到对应的使用场景，而且这个类的api设计的也有些奇怪，只有一个public abstract view

```
getViewForPositionAndType(@NotNull RecyclerView recycler, int position, int type);
```

让开发者重写通过position和type拿到ViewHolder的方法，却没有提供如何产生ViewHolder或者管理ViewHolder的方法，给人一种只出不进的赶脚，还是那句话慎用。

RecycledViewPool: 刚才说了Cache默认的缓存数量是2个，当Cache缓存满了以后会根据FIFO（先进先出）的规则把Cache先缓存进去的ViewHolder移出并缓存到RecycledViewPool中，RecycledViewPool默认的缓存数量是5个。RecycledViewPool与Cache相比不同的是，从Cache

里面移出的ViewHolder再存入RecycledViewPool之前ViewHolder的数据会被全部重置，相当于一个新的ViewHolder，而且Cache是根据position来获取ViewHolder，而**RecycledViewPool是根据itemType获取的**，如果没有重写getItemType()方法，itemType就是默认的。因为RecycledViewPool缓存的ViewHolder是全新的，所以取出来的时候需要走**onBindViewHolder()**方法。

ref

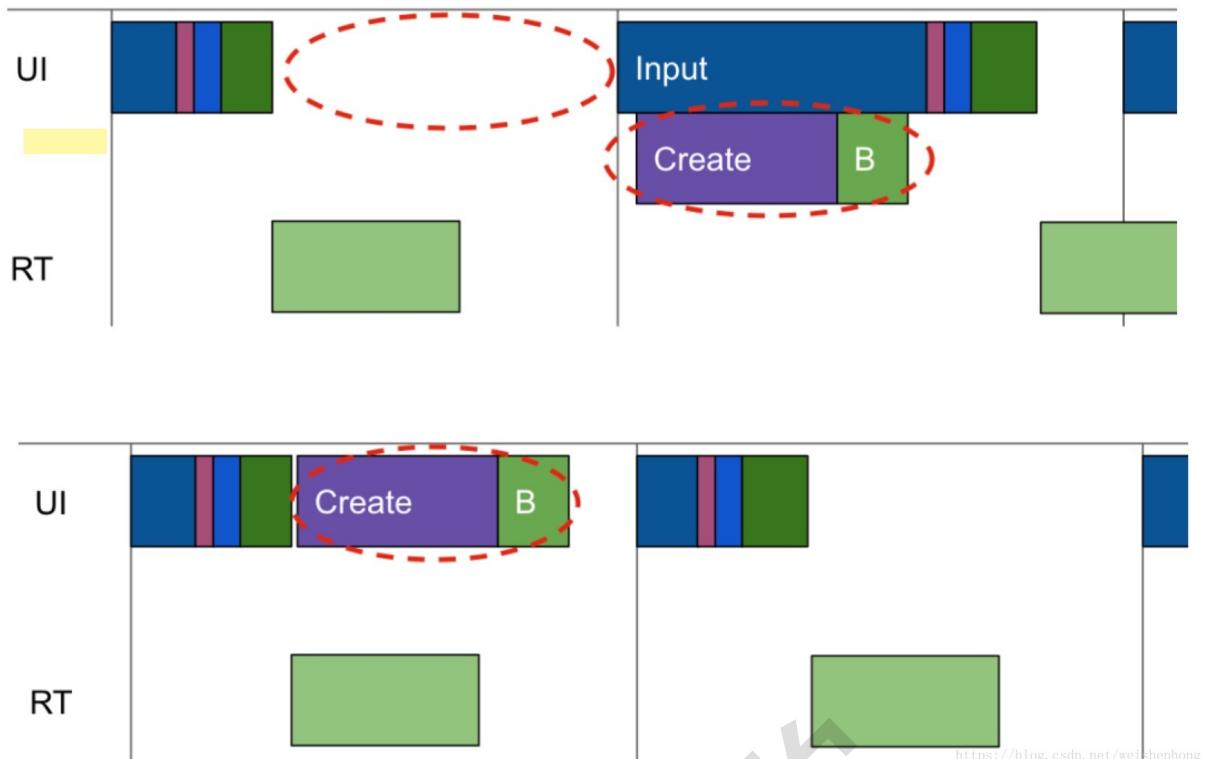
[让你彻底掌握RecyclerView的缓存机制](#)

Prefetch功能的使用

google官方在 Support Library v25 版本中，为RecyclerView增加了Prefetch。并且在 v25.1.0 以及25.3.0版本中进行了完善。在最新的稳定版本25.3.1中已经基本稳定。

Prefetch 默认就是处理开启的状态，通过LinearLayoutManager的setItemPrefetchEnabled()我们可以手动控制该功能的开启关闭。

我们都知道android是通过每16ms刷新一次页面来保证ui的流畅程度，现在android系统中刷新ui会通过cpu产生数据，然后交给gpu渲染的形式来完成，从上图可以看出当cpu完成数据处理交给gpu后就一直处于空闲状态，需要等待下一帧才会进行数据处理，而这空闲时间就被白白浪费了，如何才能压榨cpu的性能，让它一直处于忙碌状态，这就是rv的预取功能(Prefetch)要做的事情，rv会预取接下来可能要显示的item，在下一帧到来之前提前处理完数据，然后将得到的itemholder缓存起来，等到真正要使用的时候直接从缓存取出来即可。



<https://blog.csdn.net/weishenhang>

预取代码理解

虽说预取是默认开启不需要我们开发者操心的事情，但是明白原理还是能加深该功能的理解。下面就说下自己在看预取源码时的一点理解。实现预取功能的一个关键类就是gapworker，可以直接在rv源码中找到该类

```
GapWorker mGapWorker;
```

rv通过在ontouchevent中触发预取的判断逻辑，在手指执行move操作的代码末尾有这么段代码

```
case MotionEvent.ACTION_MOVE: {
    ...
    if (mGapWorker != null && (dx != 0 || dy != 0)) {
        mGapWorker.postFromTraversal(this, dx, dy);
    }
}
```

通过每次move操作来判断是否预取下一个可能要显示的item数据，判断的依据就是通过传入的dx和dy得到手指接下来可能要移动的方向，如果dx或者dy的偏移量会导致下一个item要被显示出来则预取出来，但是并不是说预取下一个可能要显示的item一定都是成功的，其实每次rv取出要显示的一个item本质上就是取出一个viewholder，根据viewholder上关联的itemview来展示这个item。而取出viewholder最核心的方法就是

```
tryGetViewHolderForPositionByDeadline(int  
position, boolean dryRun, long deadlineNs)
```

名字是不是有点长，在rv源码中你会时不时见到这种巨长的方法名，看方法的参数也能找到和预取有关的信息，deadlineNs的一般取值有两种，一种是为了兼容版本25之前没有预取机制的情况，兼容25之前的参数为

```
static final long FOREVER_NS = Long.MAX_VALUE;
```

，另一种就是实际的deadline数值，超过这个deadline则表示预取失败，这个其实也好理解，预取机制的主要目的就是提高rv整体滑动的流畅性，如果要预取的viewholder会造成下一帧显示卡顿强行预取的话那就有点本末倒置了。

关于预取成功的条件通过调用

```
boolean willCreateInTime(int viewType, long  
approxCurrentNs, long deadlineNs) {  
    long expectedDurationNs =  
        getScrapDataForType(viewType).mCreateRunningAverageNs;  
    return expectedDurationNs == 0 ||  
(approxCurrentNs + expectedDurationNs < deadlineNs);  
}
```

来进行判断，approxCurrentNs的值为

```
long start = getNanoTime();  
if (deadlineNs != FOREVER_NS  
    &&  
    !mRecyclerPool.willCreateInTime(type, start, deadlineNs))  
{  
    // abort - we have a deadline we can't meet  
    return null;  
}
```

而mCreateRunningAverageNs就是创建同type的holder的平均时间，感兴趣的可以去看下这个值如何得到，不难理解就不贴代码了。关于预取就说到这里，感兴趣的可以自己去看下其余代码的实现方式，可以说google对于rv还是相当重视的，煞费苦心提高rv的各种性能，据说最近推出的viewpager2控件就是通过rv来实现的，大有rv控件一统天下的感觉。

预取功能的应用

预取功能是默认开启的，在对应的RecyclerView.LayoutManager中提供了一个setInitialPrefetchItemCount(int itemCount)来设置预取个数。有一种场景，比如你在垂直列表里面有一个水平滚动列表的时候，竖屏每一行都是展示三个半item，可以调用**内部**（横向）Recyclerview的innerLLM.setInitialItemsPrefetchCount(4)，这样当水平列表将要展示在屏幕上的时候，如果UI线程有空闲时间，RecyclerView会尝试在内部预先把这几个item拿出来。

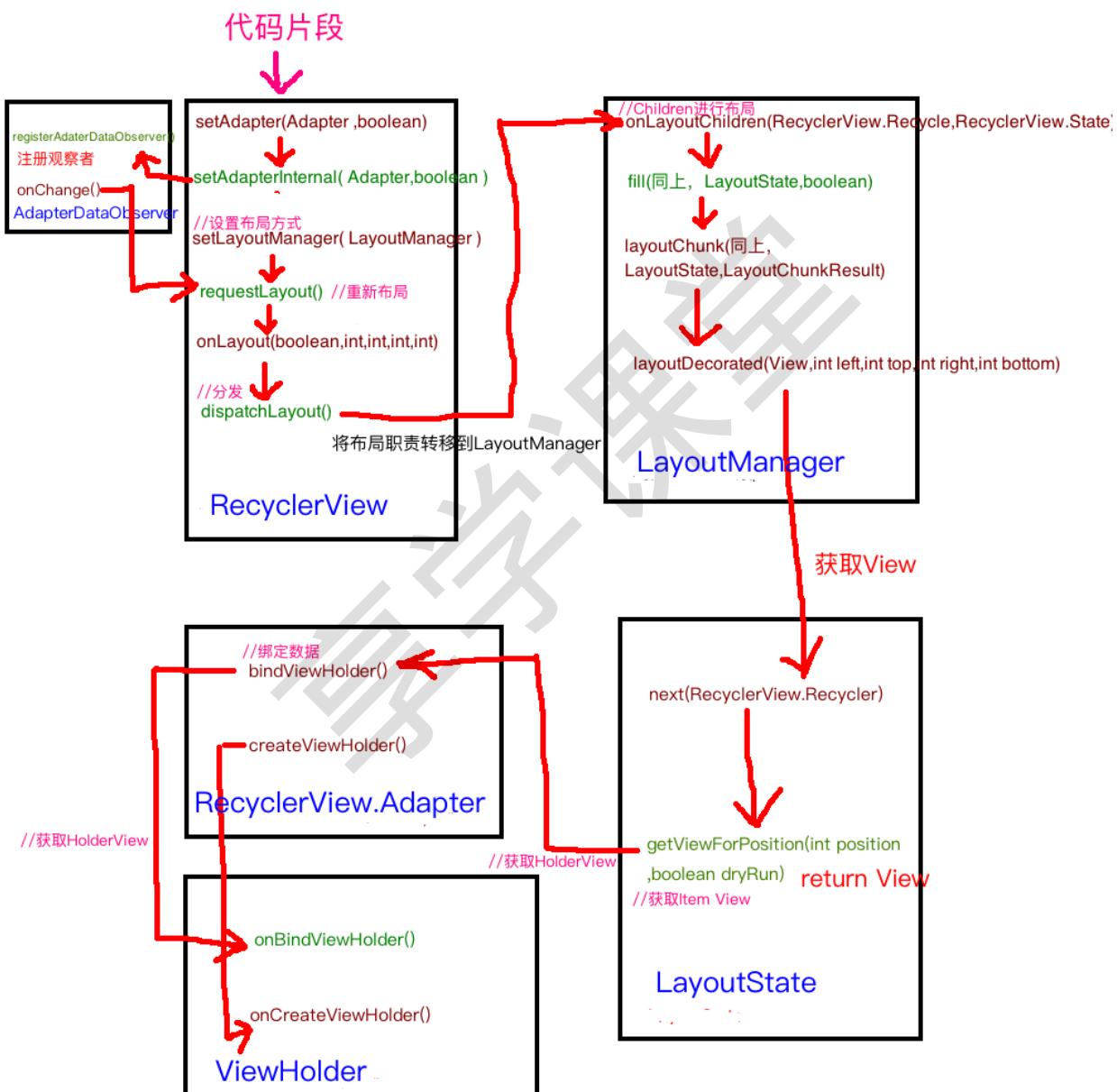
2.1.5 RecyclerView 中的设计模式

Recyclerview 在 Android 应用开发中使用非常频繁，但一般人只知道onCreateViewHolder、onBindViewHolder 等简单使用。了解这篇文章无论是对项目中实际开发还是 Android 源码设计，都会有借鉴和帮助

Recyclerview 源码中用到的设计模式

本文通过桥接, 组合, 适配器, 观察者模式四种设计模式来解读RecyclerView。

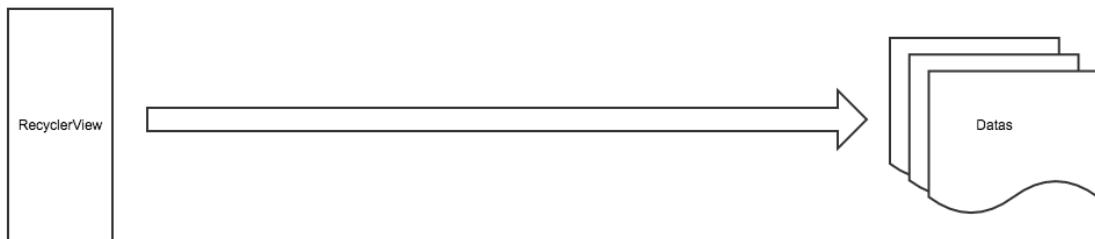
- 1. 通过**桥接模式**, 使 RecyclerView 将布局方式独立成 LayoutManager, 实现对布局的定制化。
 - 1. 通过**组合模式**, 使 RecycleView 通过 dispatchLayout 对 Item View 进行布局绘制的。
 - 1. 通过**适配器模式**, ViewHolder 将 RecycleView 与 itemView 联系起来, 使得 RecycleView 方便操作 itemView。
 - 1. 通过**观察者模式**, 给 ViewHolder 注册观察者, 当调用 notifyDataSetChanged 时, 就能重新绘制。



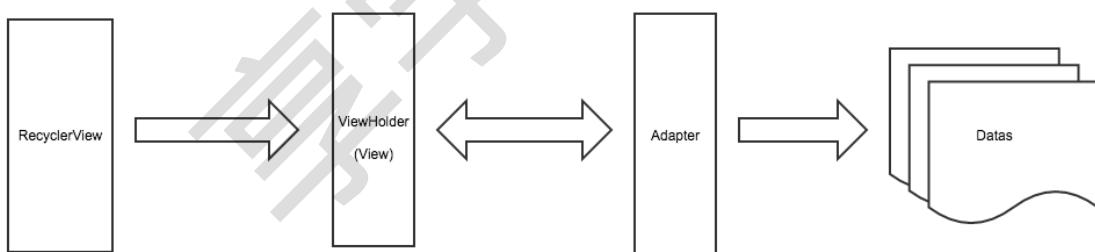
RecyclerView设计模式草图

设计思路

RecyclerView 官网给出的定义是: A flexible view for providing a limited window into a large data set. , 也就是在限定的视图内展示大量的数据, 来一张通俗明了的图:

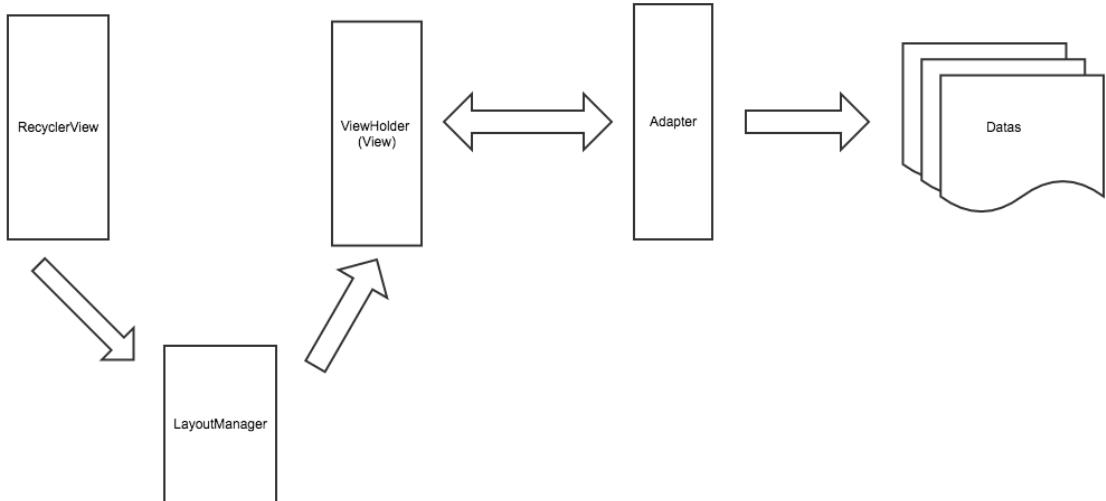


RecyclerView 的职责就是将 Datas 中的数据以一定的规则展示在它的上面, 但说破天 RecyclerView 只是一个 ViewGroup, 它只认识 View, 不清楚 Data 数据的具体结构, 所以两个陌生人之间想构建通话, 我们很容易想到适配器模式, 因此, RecyclerView 需要一个 **Adapter(适配器模式)** 来与 Datas 进行交流:

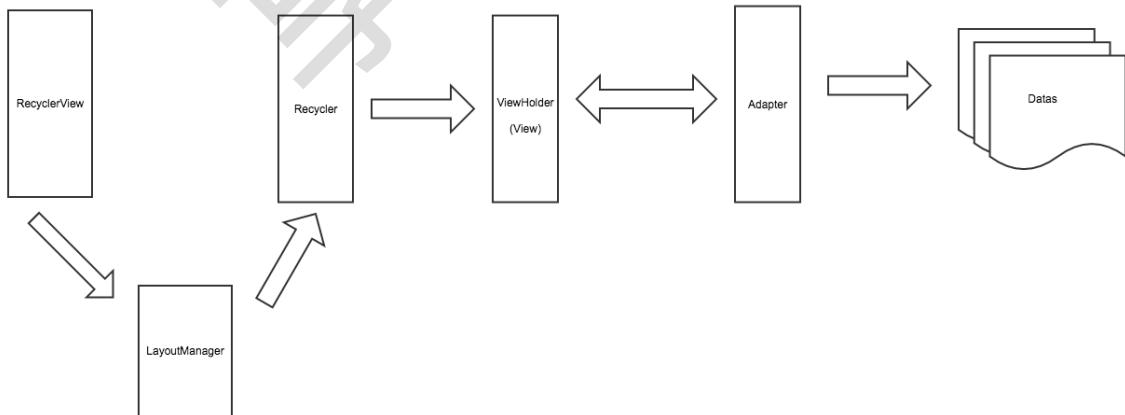


如上所示, RecyclerView 表示只会和 ViewHolder 进行接触, 而 Adapter 的工作就是将 Data 转换为 RecyclerView 认识的 ViewHolder, 因此 RecyclerView 就间接地认识了 Datas。

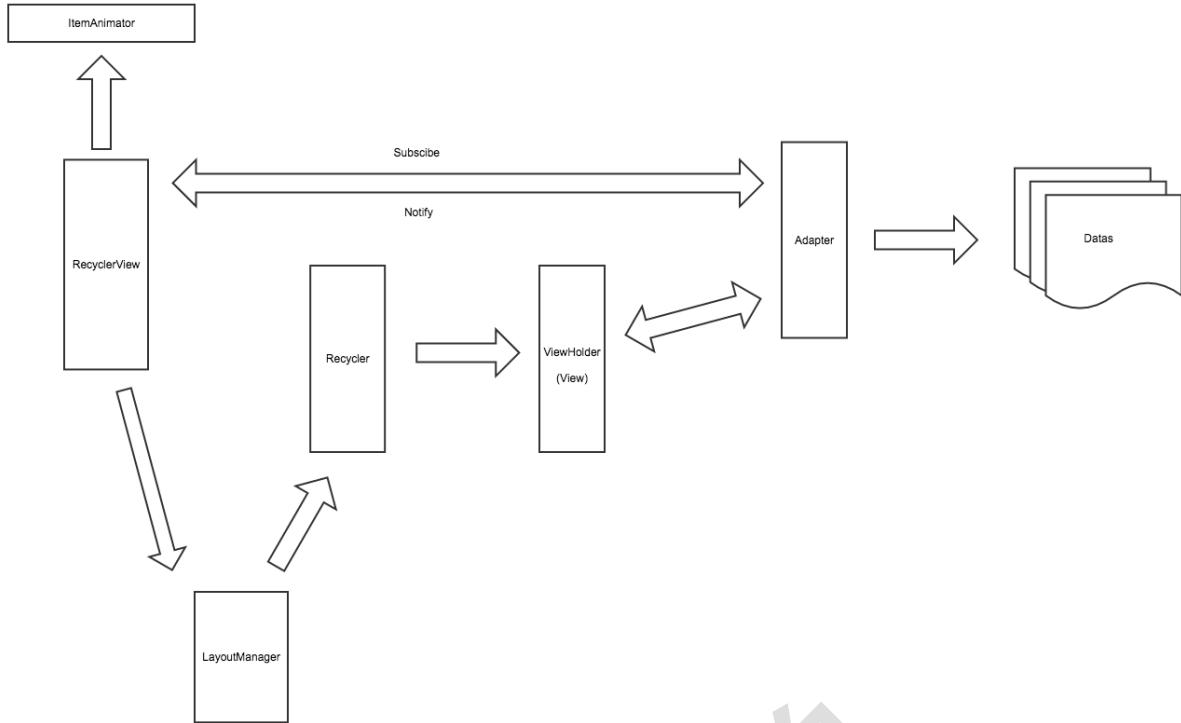
事情虽然进展愉快, 但 RecyclerView 是个很懒惰的人, 尽管 Adapter 已经将 Datas 转换为 RecyclerView 所熟知的 View, 但 RecyclerView 并不想自己管理些子 View, 因此, 它雇佣了一个叫做 **LayoutManager** 的大祭司来帮其完成布局 (桥接模式), 现在, 图示变成下面这样:



如上图所示，LayoutManager 协助 RecyclerView 来完成布局。但 LinearLayoutManager 这个大祭司也有弱点，就是它只知道如何将一个一个的 View 布局在 RecyclerView 上，但它并不懂得如何管理这些 View，如果大祭司肆无忌惮的玩弄 View 的话肯定会出现问题，所以，必须有个管理 View 的护法，它就是 **Recycler**，LayoutManager 在需要 View 的时候回向护法进行索取，当 LinearLayoutManager 不需要 View(试图滑出)的时候，就直接将废弃的 View 丢给 Recycler，图示如下：



到了这里，有负责翻译数据的 Adapter，有负责布局的 LayoutManager，有负责管理 View 的 Recycler，一切都很完美，但 RecyclerView 乃何等神也，它下令说当子 View 变动的时候姿态要优雅（动画），所以用雇佣了一个舞者 **ItemAnimator(观察者模式)**，因此，舞者也进入了这个图示：



如上，我们就是从宏观层面来对 RecyclerView 有个大致的了解，可以看到，RecyclerView 作为一个 View，它只负责接受用户的各种讯息，然后将信息各司其职的分发出去。接下来我们将深入源码，详细讲解用到的设计模式，看看 RecyclerView 都是怎么来操作各个组件工作的。

桥接模式详解

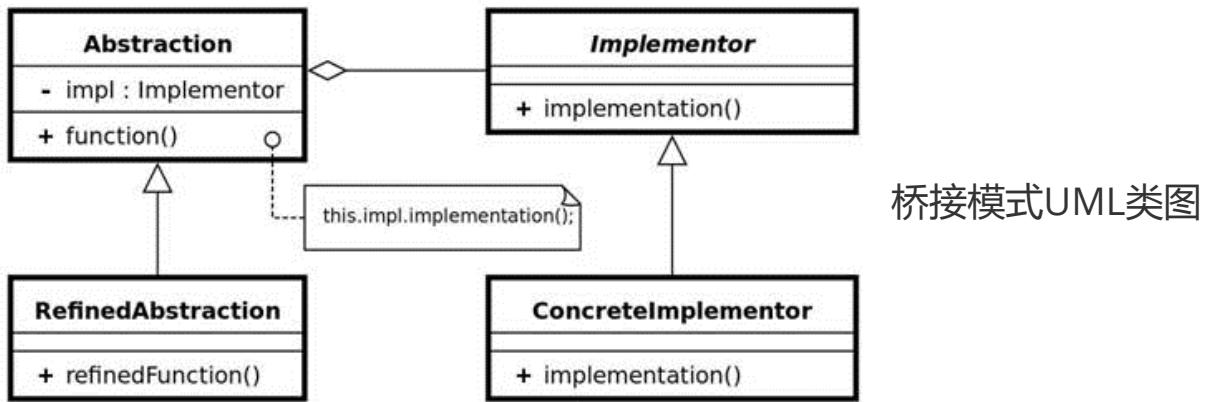
模式的定义

将抽象部分与实现部分分离，使它们都可以独立的变化。

模式的使用场景

如果一个系统需要在构件的抽象化角色和具体化角色之间添加更多的灵活性，避免在两个层次之间建立静态的联系。设计要求实现化角色的任何改变不应当影响客户端，或者实现化角色的改变对客户端是完全透明的。需要跨越多个平台的图形和窗口系统上。一个类存在两个独立变化的维度，且两个维度都需要进行扩展。

UML 类图



桥接模式UML类图

角色介绍

抽象化 (Abstraction) 角色: 抽象化给出的定义，并保存一个对实现化对象的引用。
修正抽象化 (Refined Abstraction) 角色: 扩展抽象化角色，改变和修正父类对抽象化的定义。
实现化 (Implementor) 角色: 这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。
具体实现化 (ConcretelImplementor) 角色: 这个角色给出实现化角色接口的具体实现。

模式的简单实现

其实 Java 的虚拟机就是一个很好的例子，在不同平台平台上，用不同的虚拟机进行实现，这样只需把 Java 程序编译成符合虚拟机规范的文件，且只用编译一次，便在不同平台上都能工作。但是这样说比较抽象，用一个简单的例子来实现 bridge 模式。

编写一个程序，使用两个绘图的程序的其中一个来绘制矩形或者原型，同时，在实例化矩形的时候，它要知道使用绘图程序 1 (DP1) 还是绘图程序 2 (DP2)。

(ps: 假设 dp1 和 dp2 的绘制方式不一样，它们是用不同方式进行绘制，示例代码，不讨论过多细节)

实现源码

首先是两个绘图程序 dp1,dp2

```
//具体的绘图程序类dp1
```

```
public class DP1 {  
    public void draw_1_Rantanle(){  
        System.out.println("使用DP1的程序画矩形");  
    }  
    public void draw_1_circle(){  
        System.out.println("使用DP1的程序画圆形");  
    }  
}  
//具体的绘图程序类dp2  
public class DP2 {  
    public void drawRantanle(){  
        System.out.println("使用DP2的程序画矩形");  
    }  
    public void drawCircle(){  
        System.out.println("使用DP2的程序画圆形");  
    }  
}
```

接着抽象的形状 Shape 和两个派生类：矩形 Rantanle 和圆形 Circle

```
//抽象化角色Abstraction  
abstract class Shape {  
    //持有实现的角色Implementor(作图类)  
    protected Drawing myDrawing;  
    public Shape(Drawing drawing) {  
        this.myDrawing = drawing;  
    }  
    abstract public void draw();  
    //保护方法drawRectangle  
    protected void drawRectangle(){  
        //this.impl.implemtation()  
        myDrawing.drawRantangle();  
    }  
    //保护方法drawCircle  
    protected void drawCircle(){  
        //this.impl.implemtation()
```

```
        myDrawing.drawCircle();
    }
}

//修正抽象化角色Refined Abstraction(矩形)
public class Rantangle extends Shape{
    public Rantangle(Drawing drawing) {
        super(drawing);
    }
    @Override
    public void draw() {
        drawRectangle();
    }
}

//修正抽象化角色Refined Abstraction(圆形)
public class Circle extends Shape {
    public Circle(Drawing drawing) {
        super(drawing);
    }
    @Override
    public void draw() {
        drawCircle();
    }
}
```

最后，我们的实现绘图的 Drawing 和分别实现 dp1 的 V1Drawing 和 dp2 的 V2Drawing

```
//实现化角色Implementor
//implementation两个方法，画圆和画矩形
public interface Drawing {
    public void drawRantangle();
    public void drawCircle();
}

//具体实现化逻辑ConcreteImplementor
//实现了接口方法，使用DP1进行绘图
public class V1Drawing implements Drawing{
```

```
DP1 dp1;
public V1Drawing() {
    dp1 = new DP1();
}
@Override
public void drawRantangle() {
    dp1.draw_1_Rantanle();
}
@Override
public void drawCircle() {
    dp1.draw_1_circle();
}
}

//具体实现化逻辑ConcreteImplementor
//实现了接口方法，使用DP2进行绘图
public class V2Drawing implements Drawing{
    DP2 dp2;
    public V2Drawing() {
        dp2 = new DP2();
    }
    @Override
    public void drawRantangle() {
        dp2.drawRantanle();
    }
    @Override
    public void drawCircle() {
        dp2.drawCircle();
    }
}
```

在这个示例中，图形 Shape 类有两种类型，圆形和矩形，为了使用不同的绘图程序绘制图形，把实现的部分进行了分离，构成了 Drawing 层次结构，包括 V1Drawing 和 V2Drawing。在具体实现类中，V1Drawing 控制着 DP1 程序进行绘图，V2Drawing 控制着 DP2 程序进行绘图，以及保护的方法 drawRantangle,drawCircle(Shape 类中)。

组合模式详解

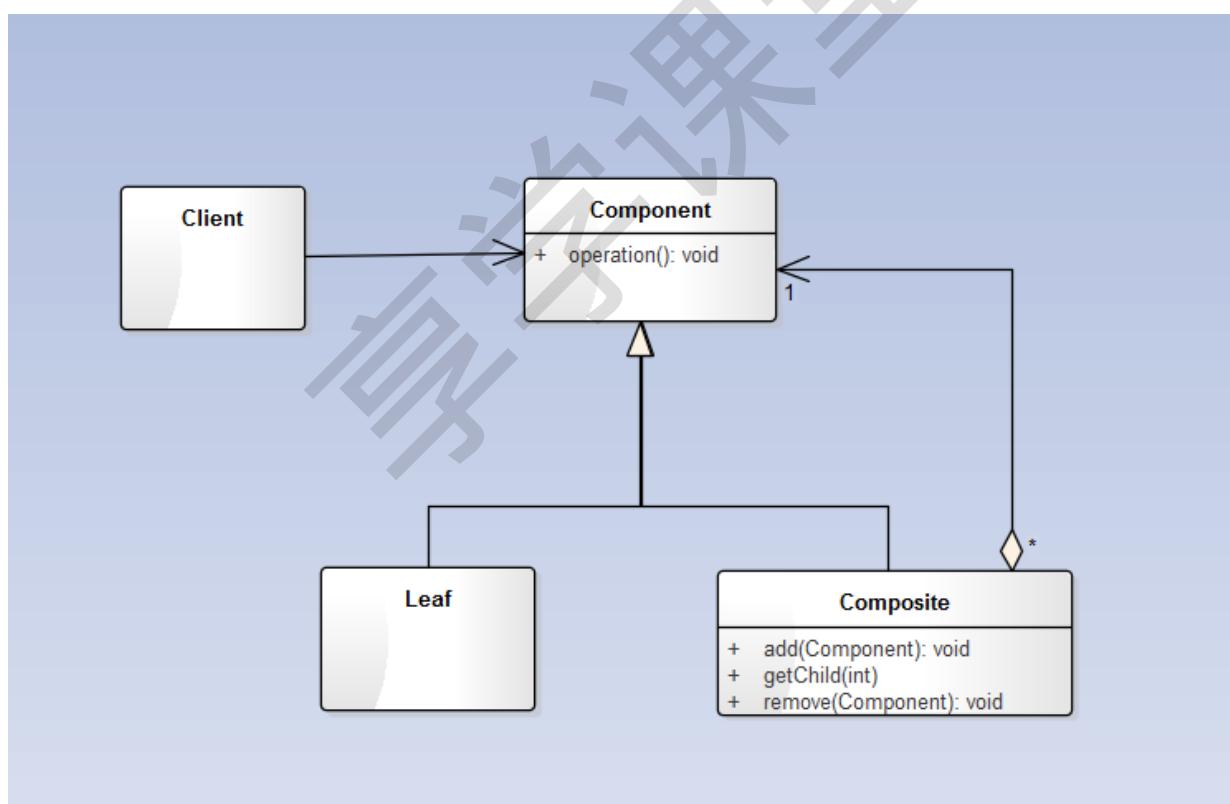
模式的定义

组合模式 (Composite Pattern) 又叫作部分 - 整体模式，它使我们在树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。GoF 在《设计模式》一书中这样定义组合模式：将对象组合成树形结构以表示“部分 - 整体”的层次结构。使得用户对单个对象和组合对象的使用具有一致性。

模式的使用场景

表示对象的部分 - 整体层次结构。从一个整体中能够独立出部分模块或功能的场景。

UML 类图



桥接模式UML类图

角色分析

Component 抽象构件角色：定义参加组合对象的共有方法和属性，可以定义一些默认的行为或属性。Leaf 叶子构件：叶子对象，其下再也没有其他的分支。Composite 树枝构件：树枝对象，它的作用是组合树枝节点和叶子节点形成一个树形结构。

该模式的实现实例

抽象构件 Component.java：

```
public abstract class Component {  
    //个体和整体都具有的共享  
    public void doSomething(){  
        //业务逻辑  
    }  
}
```

树枝构件 Composite.java

```
public class Composite extends Component {  
    //构件容器  
    private ArrayList componentArrayList = new  
ArrayList();  
    //增加一个叶子构件或树枝构件  
    public void add(Component component){  
        this.componentArrayList.add(component);  
    }  
    //删除一个叶子构件或树枝构件  
    public void remove(Component component){  
        this.componentArrayList.remove(component);  
    }  
    //获得分支下的所有叶子构件和树枝构件  
    public ArrayList getChildren(){  
        return this.componentArrayList;  
    }  
}
```

树叶构件 Leaf.java

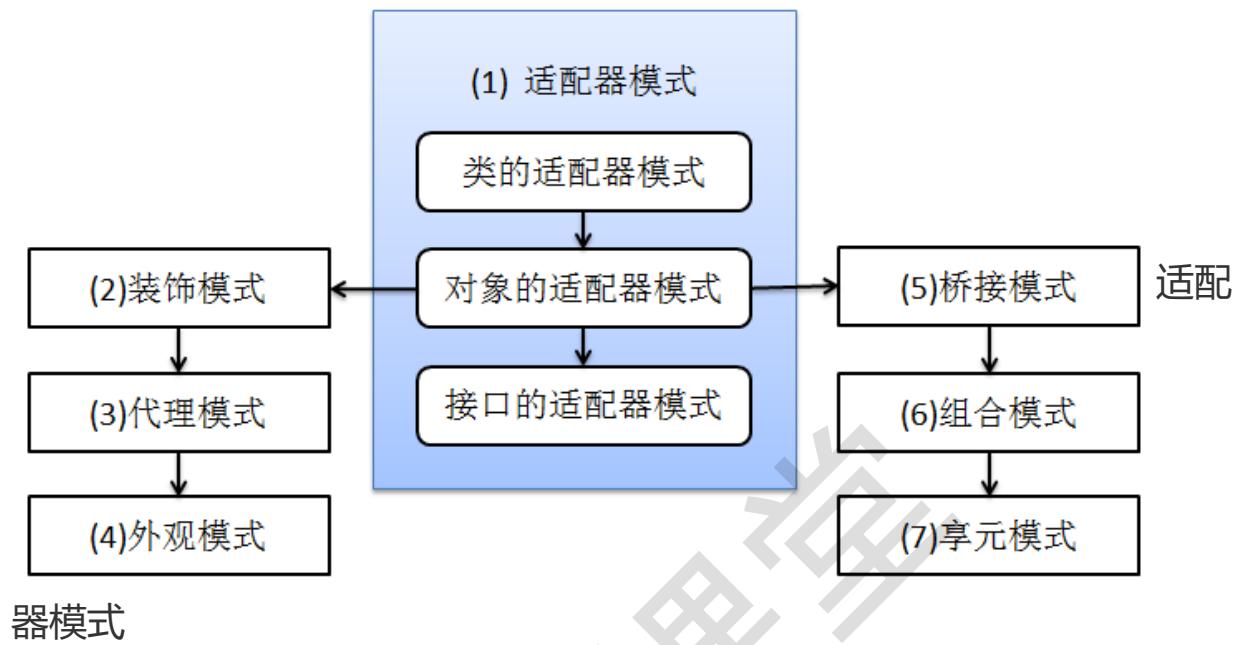
```
public class Leaf extends Component {  
    //可以覆写父类方法  
    public void doSomething(){  
    }  
}
```

场景类 Client.java

```
public class Client {  
    public static void main(String[] args) {  
        //创建一个根节点  
        Composite root = new Composite();  
        root.doSomething();  
        //创建一个树枝构件  
        Composite branch = new Composite();  
        //创建一个叶子节点  
        Leaf leaf = new Leaf();  
        //建立整体  
        root.add(branch);  
        branch.add(leaf);  
    }  
    //通过递归遍历树  
    public static void display(Composite root){  
        for(Component c:root.getChildren()){  
            if(c instanceof Leaf){ //叶子节点  
                c.doSomething();  
            }else{ //树枝节点  
                display((Composite)c);  
            }  
        }  
    }  
}
```

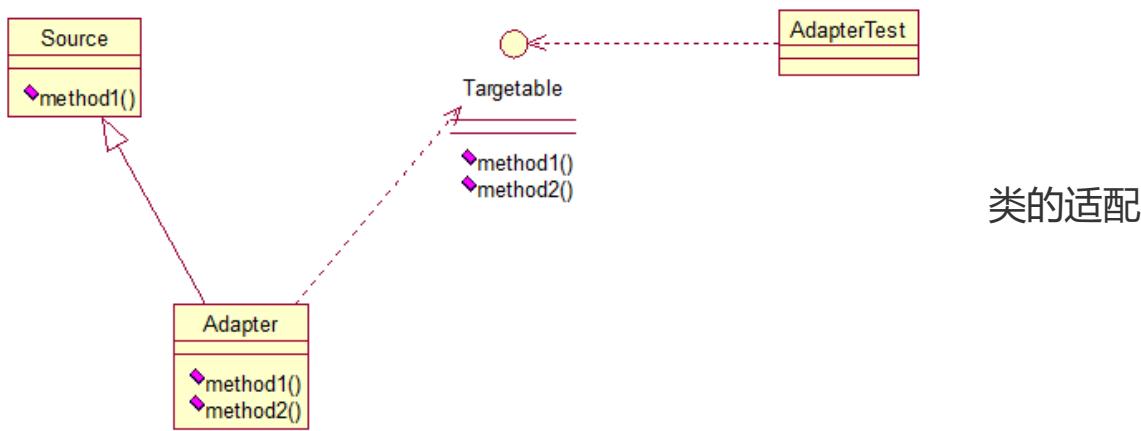
适配器模式详解

7 种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：



适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。首先，我们来看看类的适配器模式，先看类图：

类的适配器模式



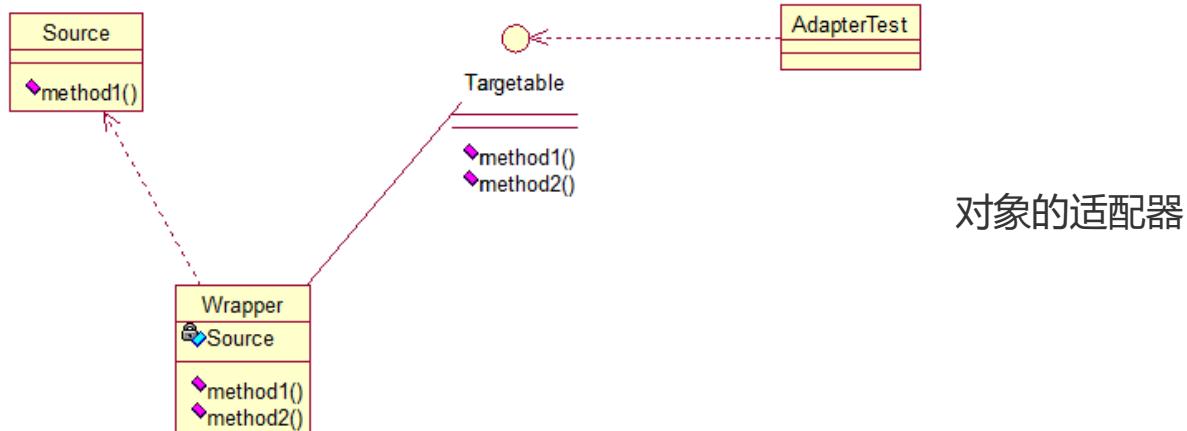
核心思想就是：有一个 Source 类，拥有一个方法，待适配，目标接口是 Targetable，通过 Adapter 类，将 Source 的功能扩展到 Targetable 里，看代码：

```
public class Source {  
    public void method1() {  
        System.out.println("this is original method!");  
    }  
}  
  
public interface Targetable {  
    /* 与原类中的方法相同 */  
    public void method1();  
    /* 新类的方法 */  
    public void method2();  
}  
  
public class Adapter extends Source implements Targetable {  
    @Override  
    public void method2() {  
        System.out.println("this is the targetable  
method!");  
    }  
}
```

Adapter 类继承 Source 类，实现 Targetable 接口，下面是测试类：

```
public class AdapterTest {  
    public static void main(String[] args) {  
        Targetable target = new Adapter();  
        target.method1();  
        target.method2();  
    }  
}
```

基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题。看图：



模式

只需要修改 Adapter 类的源码即可：

```
public class Wrapper implements Targetable {  
    private Source source;  
    public Wrapper(Source source){  
        super();  
        this.source = source;  
    }  
    @Override  
    public void method2() {  
        System.out.println("this is the targetable  
method!");  
    }  
    @Override  
    public void method1() {  
        source.method1();  
    }  
}
```

测试类：

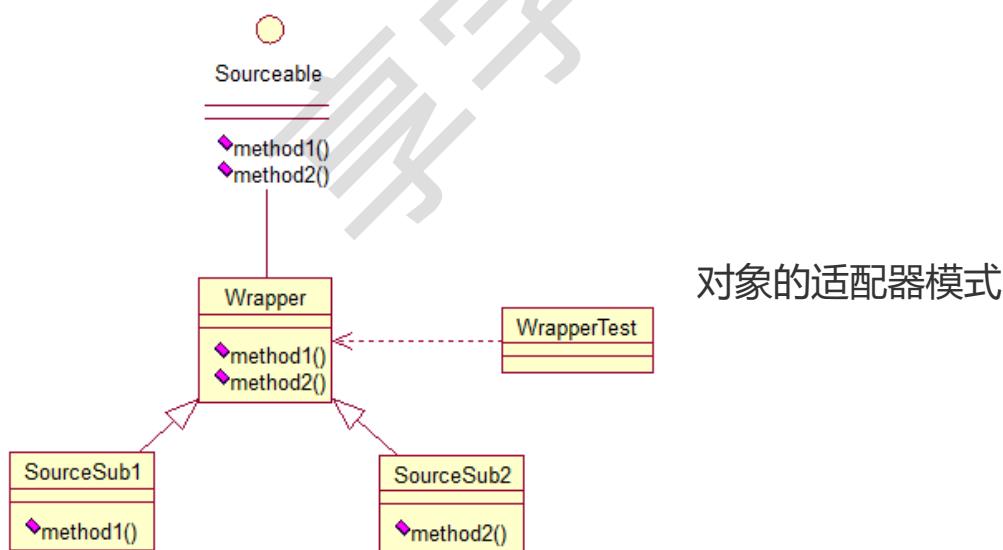
```

public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}

```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

```
public interface Sourceable {  
    public void method1();  
    public void method2();  
}
```

抽象类 Wrapper2:

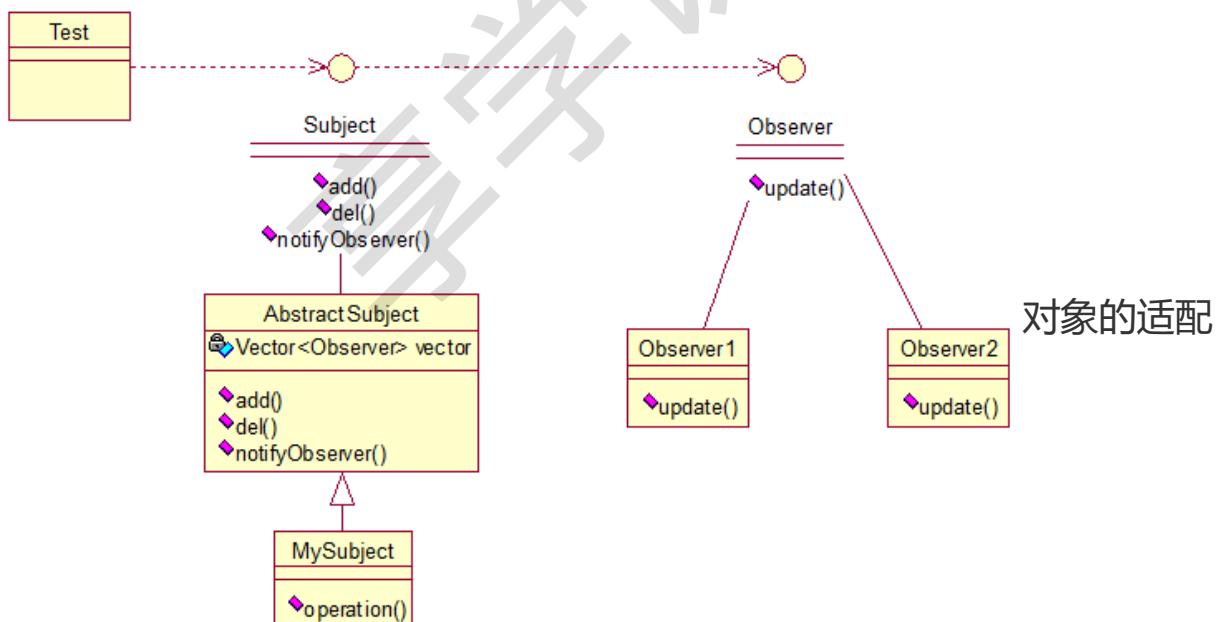
```
public abstract class Wrapper2 implements Sourceable{  
    public void method1(){}
    public void method2(){}
}  
  
public class SourceSub1 extends Wrapper2 {  
    public void method1(){  
        System.out.println("the sourceable interface's  
first Sub1!");  
    }
}  
  
public class SourceSub2 extends Wrapper2 {  
    public void method2(){  
        System.out.println("the sourceable interface's  
second Sub2!");  
    }
}  
  
public class WrapperTest {  
    public static void main(String[] args) {  
        Sourceable source1 = new SourceSub1();  
        Sourceable source2 = new SourceSub2();  
        source1.method1();  
        source1.method2();  
        source2.method1();  
        source2.method2();
    }
}
```

讲了这么多，总结一下三种适配器模式的应用场景：

- 类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。
- 对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个 Wrapper 类，持有原类的一个实例，在 Wrapper 类的方法中，调用实例的方法就行。
- 接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类 Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

观察者模式详解

观察者模式（Observer）是类和类之间的关系，不涉及到继承，学的时候应该记得归纳。观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。先来看看关系图：



器模式

我解释下这些类的作用：**MySubject** 类就是我们的主对象，**Observer1** 和 **Observer2** 是依赖于 **MySubject** 的对象，当 **MySubject** 变化时，**Observer1** 和 **Observer2** 必然变化。**AbstractSubject** 类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当 **MySubject** 变化时，负责通知在列表内存在的对象。我们看实现代码：

一个 Observer 接口：

```
public interface Observer {  
    public void update();  
}
```

两个实现类：

```
public class Observer1 implements Observer {  
    @Override  
    public void update() {  
        System.out.println("observer1 has received!");  
    }  
}  
  
public class Observer2 implements Observer {  
    @Override  
    public void update() {  
        System.out.println("observer2 has received!");  
    }  
}
```

Subject 接口及实现类：

```
public interface Subject {  
    /*增加观察者*/  
    public void add(Observer observer);  
    /*删除观察者*/  
    public void del(Observer observer);  
    /*通知所有的观察者*/  
    public void notifyObservers();  
    /*自身的操作*/  
    public void operation();  
}  
  
public abstract class AbstractSubject implements Subject {
```

```
private Vector vector = new Vector();
@Override
public void add(Observer observer) {
    vector.add(observer);
}
@Override
public void del(Observer observer) {
    vector.remove(observer);
}
@Override
public void notifyObservers() {
    Enumeration enumo = vector.elements();
    while(enumo.hasMoreElements()){
        enumo.nextElement().update();
    }
}
public class MySubject extends AbstractSubject {
    @Override
    public void operation() {
        System.out.println("update self!");
        notifyObservers();
    }
}
```

测试类：

```
public class ObserverTest {
    public static void main(String[] args) {
        Subject sub = new MySubject();
        sub.add(new Observer1());
        sub.add(new Observer2());
        sub.operation();
    }
}
```

这些东西，其实不难，只是有些抽象，不太容易整体理解，还有就是门面模式，适配器模式，桥接模式有些同学觉得都差不多，其实大不一样，但都是结构型模式。

- Facade (门面模式) 出现较多是这样的情况, 你有一个复杂的系统, 对应了各种情况, 客户看了说功能不错, 但是使用太麻烦. 你说没问题, 我给你提供一个统一的门面. 所以 Facade 使用的场合多是对系统的" 优化".
- Adapter 出现是这样的情况, 你有一个类提供接口 A, 但是你的客户需要一个实现接口 B 的类, 这个时候你可以写一个 Adapter 让把 A 接口变成 B 接口, 所以 Adapter 使用的场合是指鹿为马. 就是你受夹板气的时候, 一边告诉你我只能提供给你 A(鹿), 一边告诉你说我只要 B(马), 他长了四条腿, 你没办法了, 把鹿牵过去说, 这是马, 你看他有四条腿.(当然实现指鹿为马也有两种方法, 一个方法是你只露出鹿的四条腿, 说你看这是马, 这种方式就是封装方式的 Adapter 实现, 另一种方式是你把鹿牵过去, 但是首先介绍给他说这是马, 因为他长了四条腿这种是继承的方式.)
- Bridge 在一般的开发中出现的情况并不多, AWT 是一个, SWT 也算部分是, 如果你的客户要求你开发一个系统, 这个系统在 Windows 下运行界面的样子是 Windows 的样子. 在 Linux 下运行是 Linux 下的样子. 在 Macintosh 下运行要是 Mac Os 的样子. 怎么办? 定义一系列的控件 Button,Text,radio,checkBox 等等. 供上层开发者使用, 他们使用这些控件的方法, 利用这些控件构造一个系统的 GUI, 然后你为这些控件写好 Linux 的实现, 让它在 Linux 上调用 Linux 本地的对应控件, 在 Windows 上调用 Windows 本地的对应控件, 在 Macintosh 上调用 Macintosh 本地的对应控件 ok, 你的任务完成了.

2.1.6 Recyclerview 滑动相关功能总结

本文总结一下 RecyclerView 提供的滑动相关的 api，并探究一下为什么有的滑动方法不会回调监听 `onScrollStateChanged(int state)`

RecyclerView scroll 相关 API 有：

- `scrollTo(int x, int y)`

- scrollBy(int x, int y)
- scrollToPosition(int position)
- smoothScrollBy(@Px int dx, @Px int dy) 及其重载方法
- smoothScrollToPosition(int position)
- 当然，还有手势滑动

RecyclerView 通过 `addOnscrollListener(onscrollListener)` 监听滑动事件

RecyclerView 调用这个方法通知 layoutManager, scroll 状态已经改变

```
@Override  
public void onscrollStateChanged(int state) {  
    super.onscrollStateChanged(recyclerView, newState);  
    if (state == SCROLL_STATE_IDLE) {  
        //滑动停止  
    }  
}  
  
@Override  
public void onScrolled(RecyclerView recyclerView, int dx,  
int dy) {  
    super.onScrolled(recyclerView, dx, dy);  
    //...  
}
```

但是并不是所有的滑动行为都会回调到 `onScrollStateChanged`，下面来一一分析下每个滑动场景

scrollTo 和 scrollBy

`scrollTo` 和 `scrollBy` 是 View 相关方法，实现的是 `View` 内容的滑动，效果是 View 控件本身并没有滑动，而是控件上绘制的内容在控件范围内发生滑动。

看下在 RecyclerView 里面的实现：

```
// RecyclerView
```

```
@Override
public void scrollTo(int x, int y) {
    // Recyclerview 不支持滚动到绝对位置，尝试使用
    scrollToPosition 替换
    Log.w(TAG, "RecyclerView does not support scrolling
to an absolute position. "
        + "Use scrollToPosition instead");
}

@Override
public void scrollBy(int x, int y) {
    // 这个方法可以配合实现 NestedScrollingChild 的控件联动
    if (mLayout == null) {
        Log.e(TAG, "Cannot scroll without a LayoutManager
set. "
            + "Call setLayoutManager with a non-null
argument.");
        return;
    }
    if (mLayoutSuppressed) {
        return;
    }
    final boolean canScrollHorizontal =
mLayout.canScrollHorizontally();
    final boolean canScrollVertical =
mLayout.canScrollVertically();
    if (canScrollHorizontal || canScrollVertical) {
        scrollByInternal(canScrollHorizontal ? x : 0,
canScrollVertical ? y : 0, null);
    }
}
```

scrollByInternal 介绍

scrollByInternal 后面在处理手势拖动的时候也会用到，继续往下看：

```
// Recyclerview
```

```
boolean scrollByInternal(int x, int y, MotionEvent ev) {
    int unconsumedX = 0;
    int unconsumedY = 0;
    int consumedX = 0;
    int consumedY = 0;

    consumePendingUpdateOperations();
    if (mAdapter != null) {
        mReusableIntPair[0] = 0;
        mReusableIntPair[1] = 0;
        // 这里处理滑动
        scrollStep(x, y, mReusableIntPair);
        consumedX = mReusableIntPair[0];
        consumedY = mReusableIntPair[1];
        unconsumedX = x - consumedX;
        unconsumedY = y - consumedY;
    }
    // ... 分发 nestedscroll 结果
    dispatchNestedScroll(consumedX, consumedY,
unconsumedX, unconsumedY, mScrollOffset,
        TYPE_TOUCH, mReusableIntPair);
    // ... 通知 onScrolled 事件
    if (consumedX != 0 || consumedY != 0) {
        dispatchOnScrolled(consumedX, consumedY);
    }
    // ...
}
```

dispatchOnScrolled 会通知 onScrolled 的监听者

scrollStep 方法，如下：

```
// Recyclerview
void scrollStep(int dx, int dy, @Nullable int[] consumed)
{
    // ...
    int consumedX = 0;
    int consumedY = 0;
    if (dx != 0) { // 调用LayoutManager处理滑动
        consumedX = mLayout.scrollHorizontallyBy(dx,
mRecycler, mState);
    }
    if (dy != 0) {
        consumedY = mLayout.scrollVerticallyBy(dy,
mRecycler, mState);
    }
    // ...
}
```

LayoutManager.scrollHorizontallyBy 介绍

本文以下都以 LinearLayoutManager 为例，继续看 scrollHorizontallyBy 方法：

```
// LinearLayoutManager
int scrollBy(int delta, RecyclerView.Recycler recycler,
RecyclerView.State state) {
    if (getChildCount() == 0 || delta == 0) {
        return 0;
    }
    ensureLayoutState();
    mLayoutState.mRecycle = true;
    final int layoutDirection = delta > 0 ?
LayoutState.LAYOUT_END : LayoutState.LAYOUT_START;
    final int absDelta = Math.abs(delta);
    updateLayoutState(layoutDirection, absDelta, true,
state);
    final int consumed = mLayoutState.mScrollingOffset
```

```
        + fill(recycler, mLayoutState, state, false);
    // ...
    final int scrolled = absDelta > consumed ?
layoutDirection * consumed : delta;
    mOrientationHelper.offsetChildren(-scrolled);

    mLayoutState.mLastScrollDelta = scrolled;
    return scrolled;
}
```

mOrientationHelper 是 LinearLayoutManager 类的一个全局变量，初始化是在：

```
// LinearLayoutManager
public void setOrientation(@RecyclerView.Orientation int
orientation) {
    if (orientation != HORIZONTAL && orientation != VERTICAL) {
        throw new IllegalArgumentException("invalid orientation:" + orientation);
    }

    assertNotInLayoutOrScroll(null);

    if (orientation != mOrientation || mOrientationHelper == null) {
        mOrientationHelper =
OrientationHelper.createOrientationHelper(this,
orientation);
        mAnchorInfo.mOrientationHelper =
mOrientationHelper;
        mOrientation = orientation;
        requestLayout();
    }
}
```

继续往下看：

```
// abstract class OrientationHelper
public static OrientationHelper createOrientationHelper(
    RecyclerView.LayoutManager layoutManager,
    @RecyclerView.Orientation int orientation) {
    switch (orientation) {
        case HORIZONTAL:
            return createHorizontalHelper(layoutManager);
        case VERTICAL:
            return createVerticalHelper(layoutManager);
    }
    throw new IllegalArgumentException("invalid
orientation");
}
```

以 HORIZONTAL 方向为例：

```
public static OrientationHelper createHorizontalHelper(
    RecyclerView.LayoutManager layoutManager) {
    return new OrientationHelper(layoutManager) {
        @Override
        public int getEndAfterPadding() {
            return mLayoutManager.getWidth() -
mLayoutManager.getRightPadding();
        }

        @Override
        public int getEnd() {
            return mLayoutManager.getWidth();
        }

        @Override
        public void offsetChildren(int amount) {
            // 最终还是调用的全局变量 mLayoutManager 的
offsetChildrenHorizontal 方法
        }
    };
}
```

```
mLayoutManager.offsetChildrenHorizontal(amount);
        // 垂直方向调用的就是下面的方法:
        //
mLayoutManager.offsetChildrenVertical(amount);
    }
    //...
}
}

private OrientationHelper(RecyclerView.LayoutManager
layoutManager) {
    mLayoutManager = layoutManager;
}
```

mLayoutManager 作为 OrientationHelper 构造方法的唯一参数。
点击 mLayoutManager.offsetChildrenHorizontal(amount) 方法，跳转到的是 RecyclerView.LayoutManger 的方法：

```
// RecyclerView.LayoutManger
public void offsetChildrenHorizontal(@Px int dx) {
    if (mRecyclerView != null) {
        mRecyclerView.offsetChildrenHorizontal(dx);
    }
}
```

回到 RecyclerView 的这个方法：

```
public void offsetChildrenHorizontal(@Px int dx) {  
    final int childCount = mChildHelper.getChildCount();  
    for (int i = 0; i < childCount; i++) {  
        // getChildAt返回的是每个child view  
  
        mChildHelper.getChildAt(i).offsetLeftAndRight(dx);  
    }  
}
```

View.offsetTopAndRight 介绍

经过上面的跳转步骤，最终发现会调用到列表里每个 View 的 offsetLeftAndRight 方法：

```
// View  
public void offsetLeftAndRight(int offset) {  
    if (offset != 0) {  
        final boolean matrixIsIdentity =  
hasIdentityMatrix();  
        if (matrixIsIdentity) {  
            if (isHardwareAccelerated()) {  
                invalidateViewProperty(false, false);  
            } else {  
                final ViewParent p = mParent;  
                if (p != null && mAttachInfo != null) {  
                    final Rect r =  
mAttachInfo.mTmpInvalRect;  
                    int minLeft;  
                    int maxRight;  
                    if (offset < 0) {  
                        minLeft = mLeft + offset;  
                        maxRight = mRight;  
                    } else {  
                        minLeft = mLeft;  
                        maxRight = mRight + offset;  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        r.set(0, 0, maxRight - minLeft,
mBottom - mTop);
                p.invalidateChild(this, r);
            }
        }
    } else {
        invalidateViewProperty(false, false);
    }

    mLeft += offset;
    mRight += offset;
    mRenderNode.offsetLeftAndRight(offset);
    if (isHardwareAccelerated()) {
        invalidateViewProperty(false, false);

invalidateParentIfNeededAndWasQuickRejected();
    } else {
        if (!matrixIsIdentity) {
            invalidateViewProperty(false, true);
        }
        invalidateParentIfNeeded();
    }
    notifySubtreeAccessibilityStateChangedIfNeeded();
}
}
```

这里 RecyclerView 的每个子 View 都是通过改变子 View 的 mLeft、mTop 等坐标，将初始位置值及偏移量传入，即需要滑动到的位置的坐标完成滑动。

这里的 mLeft、mTop 是指基于**父控件**的视图坐标系中的坐标

Android 中有两种坐标系：

- 一种是视图坐标系，以当前控件左上角为坐标原点，向右为 X 轴正方向，向下为 Y 轴正方向，MotionEvent 的 getX()、getY() 方法获取的是点击位置在视图坐标系中的坐标。

- 另一种是 Android 坐标系，以屏幕左上角为坐标原点，向右为 X 轴正方向，向下为 Y 轴正方向，MotionEvent 的 getRawX()、getRawY() 方法获取的是点击位置在 Android 坐标系中的坐标

小结

小结一下，scrollBy 方法通知了 onScrolled 回调，但是没有通知 onScrollStateChanged 回调

额外补充一点 RecyclerView 的布局过程，看上面 scrollBy 里调用的 fill 方法

```
// LinearLayoutManager
// fill填充方法， 返回的是填充ItemView需要的像素，以便拿去做滚动
int fill(RecyclerView.Recycler recycler, LayoutState
layoutState,
          RecyclerView.State state, boolean
stopOnFocusable) {
    // 填充起始位置
    final int start = layoutState.mAvailable;
    if (layoutState.mScrollingOffset != LayoutState.SCROLLING_OFFSET_NaN) {
        //如果有滚动就执行一次回收
        recycleByLayoutState(recycler, layoutState);
    }
    // 计算剩余可用的填充空间
    int remainingSpace = layoutState.mAvailable +
layoutState.mExtraFillSpace;
    // 用于记录每一次while循环的填充结果
    LayoutChunkResult layoutChunkResult =
mLayoutChunkResult;

    // ===== 核心while循环
=====

    while ((layoutState.mInfinite || remainingSpace > 0)
&& layoutState.hasMore(state)) {
```

```
// ===== 填充itemView核心填充方法 ===== 屏幕还有剩余  
可用空间并且还有数据就继续执行  
    layoutChunk(recycler, state, layoutState,  
    layoutChunkResult);  
    // ...  
}  
// 填充完成后修改起始位置  
return start - layoutState.mAvailable;  
}
```

这个方法是用来填充内容的，更多布局过程可以参考这篇文章：[《图文详解 LinearLayoutManager 填充、测量、布局过程》](#)

scrollToPosition

再来看下 scrollToPosition，这个方法会使 RecyclerView 滚动最小的距离，以使目标位置可见，如果目标位置的 view 没有创建，则滑动不会发生

```
// Recyclerview.java  
public void scrollToPosition(int position) {  
    if (mLayoutSuppressed) {  
        return;  
    }  
    stopScroll();  
    if (mLayout == null) {  
        Log.e(TAG, "Cannot scroll to position a  
LayoutManager set. "  
                + "Call setLayoutManager with a non-null  
argument.");  
        return;  
    }  
    mLayout.scrollToPosition(position); // 实际调用的是  
    LayoutManager的对应方法  
    awakenScrollBars();  
}
```

这个方法调用了 LayoutManager 的同名方法：

```
// LinearLayoutManager.java
@Override
public void scrollToPosition(int position) {
    // 用 mPendingScrollPosition 变量保存方法传入的位置
    mPendingScrollPosition = position;
    mPendingScrollPositionOffset = INVALID_OFFSET;
    if (mPendingSavedState != null) {
        mPendingSavedState.invalidateAnchor();
    }
    requestLayout(); // 请求刷新布局
}
```

requestLayout(), 会重走 onMeasure, onLayout 过程, 在 RecyclerView 的 dispatchLayoutStep1() 中会调用 onLayoutChildren() 方法。

同时 LinearLayoutManager 重写了 onLayoutChildren 方法

```
// LinearLayoutManager.java
@Override
public void onLayoutChildren(RecyclerView.Recycler
recycler, RecyclerView.State state) {
    // layout algorithm: 布局算法
    // 1) by checking children and other variables, find
    an anchor coordinate and an anchor item position.
    // 通过检查孩子和其他变量, 找到锚坐标和锚点项目位置    mAnchor
    为布局锚点 理解为不具有的起点.
    // mAnchor包含了子控件在Y轴上起始绘制偏移量
    (coordinate), ItemView在Adapter中的索引位置(position)和布局方
    向(mLayoutFromEnd)
    // 2) fill towards start, stacking from bottom 开始填
    充, 从底部堆叠
    // 3) fill towards end, stacking from top 结束填充, 从顶
    部堆叠
    // 4) scroll to fulfill requirements like stack from
    bottom. 滚动以满足堆栈从底部的要求
```

```
// resolve layout direction 设置布局方向  
(VERTICAL/HORIZONTAL)  
    resolveShouldLayoutReverse();  
  
    // ...  
    final View focused = getFocusedChild();  
    if (!mAnchorInfo.mValid || mPendingScrollPosition !=  
RecyclerView.NO_POSITION  
        || mPendingSavedState != null) {  
        // mPendingScrollPosition 值已更新，会进到这里  
        mAnchorInfo.reset();  
        // mStackFromEnd需要我们开发者主动调用，不然一直未false  
        // VERTICAL方向为mLayoutFromEnd为false HORIZONTAL方  
向是为true  
        mAnchorInfo.mLayoutFromEnd = mShouldReverseLayout  
        ^ mStackFromEnd;  
        // 计算更新保存绘制锚点信息  
        updateAnchorInfoForLayout(recycler, state,  
        mAnchorInfo);  
        mAnchorInfo.mValid = true;  
    } else if (focused != null &&  
(mOrientationHelper.getDecoratedStart(focused)  
        >=  
mOrientationHelper.getEndAfterPadding()  
        ||  
mOrientationHelper.getDecoratedEnd(focused)  
        <=  
mOrientationHelper.getStartAfterPadding())) {  
  
    mAnchorInfo.assignFromViewAndKeepVisibleRect(focused,  
    getPosition(focused));  
}  
  
    // ... 如果 mPendingScrollPosition 有效，则会在  
    prelayout 阶段在用到这个值  
    if (state.isPreLayout() && mPendingScrollPosition !=  
RecyclerView.NO_POSITION
```

```
        && mPendingScrollPositionOffset !=  
INVALID_OFFSET) {  
    final View existing =  
findViewByPosition(mPendingScrollPosition);  
    if (existing != null) {  
        final int current;  
        final int upcomingOffset;  
        if (mShouldReverseLayout) {  
            current =  
mOrientationHelper.getEndAfterPadding()  
            -  
mOrientationHelper.getDecoratedEnd(existing);  
            upcomingOffset = current -  
mPendingScrollPositionOffset;  
        } else {  
            current =  
mOrientationHelper.getDecoratedStart(existing)  
            -  
mOrientationHelper.getStartAfterPadding();  
            upcomingOffset =  
mPendingScrollPositionOffset - current;  
        }  
        if (upcomingOffset > 0) {  
            extraForStart += upcomingOffset;  
        } else {  
            extraForEnd -= upcomingOffset;  
        }  
    }  
}  
  
// ...省略根据布局方向调用 fill 方法填充  
}  
  
// 更新 anchorInfo  
private boolean  
updateAnchorFromPendingData(RecyclerView.State state,  
AnchorInfo anchorInfo) {  
    //...
```

```
// 简单理解，如果位置大于itemCount，不会更新 anchorInfo，并且清除 mPendingScrollPosition 的值
    if (mPendingScrollPosition < 0 ||
mPendingScrollPosition >= state.getItemCount()) {
        mPendingScrollPosition =
RecyclerView.NO_POSITION;
        mPendingScrollPositionOffset = INVALID_OFFSET;
        if (DEBUG) {
            Log.e(TAG, "ignoring invalid scroll position
" + mPendingScrollPosition);
        }
        return false;
    }
    // ...
}
```

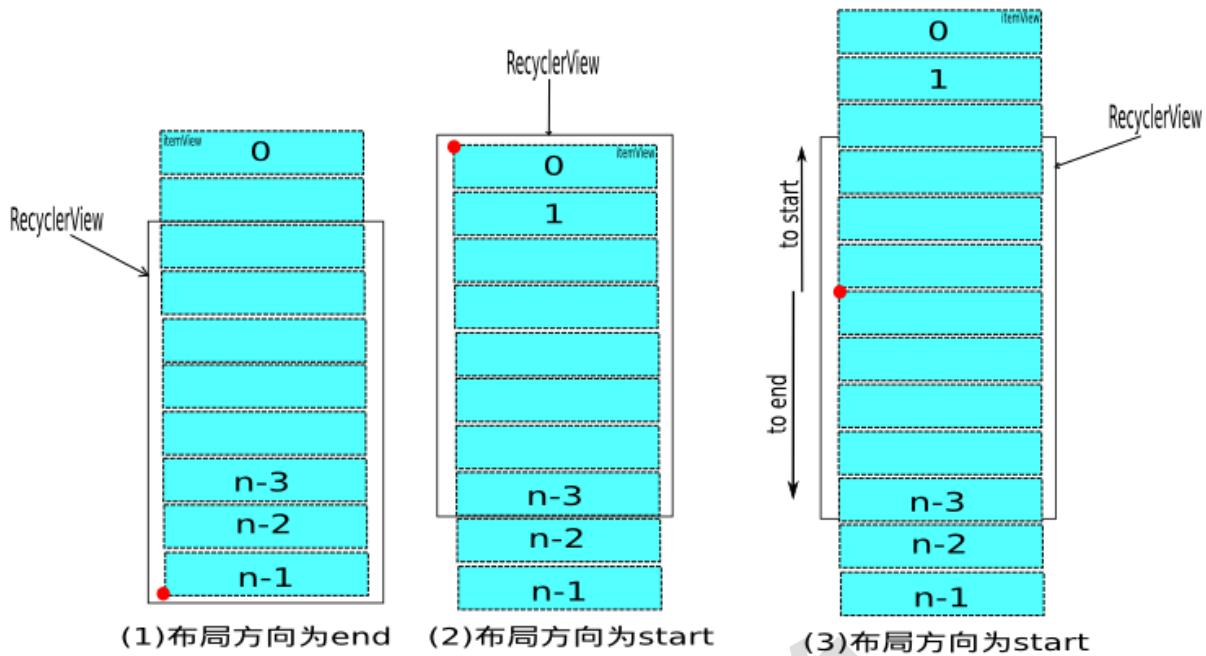
小结

上面代码可以看出，如果 scrollToPosition(position) 的参数 position 所在位置还没有 view 被 layout，则滑动不会被处理。

同时也可以看出 scrollToPosition 是 LayoutManager 通过 requestLayout 方式来刷新 RecyclerView，所以并不会通知 onScrollStateChanged

这里额外介绍一下根据 mAnchor 和布局方向填充 view 的逻辑

● 表示布局锚点



RecyclerView anchor

圆形红点就是我们布局算法在第一步 updateAnchorInfoForLayout 方法中计算出来的填充锚点位置。

- 第一种情况是屏幕显示的位置在 RecyclerView 的最底部，那么就只有一种填充方向为 formEnd
- 第二种情况是屏幕显示的位置在 RecyclerView 的顶部，那么也只有一种填充方向为 formStart
- 第三种情况应该是最常见的，屏幕显示的位置在 RecyclerView 的中间，那么填充方向就有 formEnd 和 formStart 两种情况，这就是 fill 方法调用两次的原因。

上面是 RecyclerView 的方向为 VERTICAL 的情况，当为 HORIZONTAL 方向的时候填充算法是不变的。

smoothScrollBy

smoothScrollBy 有很多重载方法，直接看最终的：

```
// RecyclerView.java
void smoothScrollBy(@Px int dx, @Px int dy, @Nullable
Interpolator interpolator,
        int duration, boolean withNestedScrolling) {
    // ... 前面都是是否能滑动判断
```

```
    if (dx != 0 || dy != 0) {
        boolean durationSuggestsAnimation = duration ==
UNDEFINED_DURATION || duration > 0;
        if (durationSuggestsAnimation) {
            if (withNestedScrolling) {
                int nestedScrollAxis =
ViewCompat.SCROLL_AXIS_NONE;
                if (dx != 0) {
                    nestedScrollAxis |=
ViewCompat.SCROLL_AXIS_HORIZONTAL;
                }
                if (dy != 0) {
                    nestedScrollAxis |=
ViewCompat.SCROLL_AXIS_VERTICAL;
                }
                startNestedScroll(nestedScrollAxis,
TYPE_NON_TOUCH);
            }
            mViewFlinger.smoothScrollBy(dx, dy, duration,
interpolator);
        } else {
            scrollBy(dx, dy);
        }
    }
}
```

最终调用了 mViewFlinger.smoothScrollBy 方法：

```
// mViewFlinger.smoothScrollBy
public void smoothScrollBy(int dx, int dy, int duration,
    @Nullable Interpolator interpolator) {

    // Handle cases where parameter values aren't
defined.
    if (duration == UNDEFINED_DURATION) {
        duration = computeScrollDuration(dx, dy, 0,
0);
    }
}
```

```
    if (interpolator == null) {
        interpolator = sQuinticInterpolator;
    }

    // If the Interpolator has changed, create a new
    // OverScroller with the new
    // interpolator.
    if (mInterpolator != interpolator) {
        mInterpolator = interpolator;
        mOverScroller = new
OverScroller(getContext(), interpolator);
    }

    // Reset the last fling information.
    mLastFlingX = mLastFlingY = 0;

    // 调用RecyclerView的setScrollState方法，通知滑动事件
开始
    setScrollState(SCROLL_STATE_SETTLING);
    mOverScroller.startScroll(0, 0, dx, dy,
duration);

    if (Build.VERSION.SDK_INT < 23) {
        mOverScroller.computeScrollOffset();
    }

    postOnAnimation();
}
```

小结

smoothScrollBy 调用了 RecyclerView.setScrollState 方法，最终会通知 onScrollStateChanged 的监听

smoothScrollToPosition

再来看下 smoothScrollToPosition

```
// Recyclerview.java
public void smoothScrollToPosition(int position) {
    if (mLayoutSuppressed) {
        return;
    }
    if (mLayout == null) {
        Log.e(TAG, "Cannot smooth scroll without a
LayoutManager set. "
                + "Call setLayoutManager with a non-null
argument.");
        return;
    }
    mLayout.smoothScrollToPosition(this, mState,
position);
}
```

上面直接调用了 LayoutManager 的同名方法

```
// LinearLayoutManager.java
@Override
public void smoothScrollToPosition(RecyclerView
recyclerView, RecyclerView.State state,
        int position) {
    LinearSmoothScroller linearSmoothScroller =
        new
LinearSmoothScroller(recyclerView.getContext());
    linearSmoothScroller.setTargetPosition(position);
    startSmoothScroll(linearSmoothScroller);
}
```

可以看到每次调用方法，都会创建一个 LinearSmoothScroller，
LinearSmoothScroller 的父类是 RecyclerView.SmoothScroller

最终调用的是 mRecyclerView.mViewFlinger.postOnAnimation();
ViewFlinger 其实一个 Runnable, 在 postOnAnimation() 内部又将该
Runnable 发送出去了
那么我们在看下 ViewFlinger 的 run() 方法就行了。

```
// Recyclerview.viewFlinger
@Override
public void run() {
    // ...
    final OverScroller scroller = mScroller;
    //获得LayoutManger中的SmoothScroller
    final SmoothScroller smoothScroller =
mLayout.mSmoothScroller;
    if (scroller.computeScrollOffset()) {//如果是第一次走,
会返回false
        // ...省略部分代码
        if (mAdapter != null) {
            mReusableIntPair[0] = 0;
            mReusableIntPair[1] = 0;
            scrollStep(unconsumedX, unconsumedY,
mReusableIntPair);
            consumedX = mReusableIntPair[0];
            consumedY = mReusableIntPair[1];
            // ...
        }
        if (consumedX != 0 || consumedY != 0) {
            dispatchOnScrolled(consumedX, consumedY);
        }
    }
    if (smoothScroller != null) {
        if (smoothScroller.isPendingInitialRun()) {
            smoothScroller.onAnimation(0, 0);
        }
        if (!mReschedulePostAnimationCallback) {
            smoothScroller.stop(); //stop if it does not
trigger any scroll
        }
    }
}
```

```
    }  
    // ...  
}
```

小结

可以看出在 ViewFlinger 的 run() 方法中，调用了 dispatchOnScrolled(consumedX, consumedY)，通知了 onScrolled()

真正产生滑动距离 consumedX、consumedY 的方法是 scrollStep()。这个方法前面分析 scrollBy() 方法的时候已经分析过了，最终会调用到每个 view 的 offsetLeftAndRight() 方法。

补充滚到一个屏幕外的位置

scrollToPosition 和 smoothScrollToPosition 只能保证指定位置的 item 滑动到屏幕可见，如果指定的 item 本来就已在屏幕可见范围，则不会滑动，并且屏幕外的 item 滑到可见范围后，还需手动置顶，手动置顶可以调用 LinearLayoutManager.scrollToPositionWithOffset(position, 0)

```
recyclerView.scrollToPosition(pos);  
linearLayoutManager.scrollToPositionWithOffset(pos, 0)
```

还可以使用另一种方式：

可以继承 LinearSmoothScroller，重写 getVerticalSnapPreference() 或 getHorizontalSnapPreference()getVerticalSnapPreference()

```
// LinearSmoothScroller
protected int getHorizontalSnapPreference() {
    return mTargetVector == null || mTargetVector.x == 0
? SNAP_TO_ANY :
    mTargetVector.x > 0 ? SNAP_TO_END :
SNAP_TO_START;
}

@Override
protected int getVerticalSnapPreference() {
    // 子view与Recyclerview垂直方向顶部对齐
    return SNAP_TO_START;
}
```

之后这样调用就可以：

```
final TopSmoothScroller mTopScroller = new
TopSmoothScroller(this);
mTopScroller.setTargetPosition(position);
mRecyclerView.getLayoutManager.startSmoothScroll(mTopScro
ller);
```

补充 scrollToPositionWithOffset

这个方式是 LinearLayoutManager 才有的方法， RecyclerView 没有。
RecyclerView 可以调用的是
LinearLayoutManger.scrollToPosition(position)

这两个方法区别是区别是 RecyclerView.scrollToPosition(position) 的
mPendingScrollPositionOffset 值为 `INVALID_OFFSET = Integer.MIN_VALUE`

```
public void scrollToPositionWithOffset(int position, int offset) {  
    mPendingScrollPosition = position;  
    mPendingScrollPositionOffset = offset;  
    if (mPendingSavedState != null) {  
        mPendingSavedState.invalidateAnchor();  
    }  
    requestLayout();  
}
```

前面也提到过了，这个方法第二个参数传 0，可以让 position 位置对应的 view 置顶

同样这个方法是 LayoutManager 通过 requestLayout 方式来刷新 RecyclerView，所以并不会通知 onScrollStateChanged。

手势滑动

手势处理肯定是在 RecyclerView 的 onTouchEvent 方法中了：

```
@Override  
public boolean onTouchEvent(MotionEvent e) {  
    switch (action) {  
        case MotionEvent.ACTION_MOVE: {  
            // ...判断手势索引，避免多指引起冲突  
            final int index =  
e.findPointerIndex(mScrollPointerId);  
  
            final int x = (int) (e.getX(index) + 0.5f);  
            final int y = (int) (e.getY(index) + 0.5f);  
            int dx = mLastTouchX - x;  
            int dy = mLastTouchY - y;  
  
            if (mScrollState != SCROLL_STATE_DRAGGING) {  
                boolean startScroll = false;  
                if (canScrollHorizontally) {  
                    if (dx > 0) {  
                        if (startScroll) {  
                            startScroll = true;  
                            scroll(dx);  
                        } else {  
                            scroll((int) ((float) dx * scrollScale));  
                        }  
                    } else if (dx < 0) {  
                        if (startScroll) {  
                            startScroll = true;  
                            scroll(-dx);  
                        } else {  
                            scroll((int) ((float) -dx * scrollScale));  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        dx = Math.max(0, dx -  
mTouchSlop);  
    } else {  
        dx = Math.min(0, dx +  
mTouchSlop);  
    }  
    if (dx != 0) {  
        startScroll = true;  
    }  
}  
if (canScrollVertically) {  
    if (dy > 0) {  
        dy = Math.max(0, dy -  
mTouchSlop);  
    } else {  
        dy = Math.min(0, dy +  
mTouchSlop);  
    }  
    if (dy != 0) {  
        startScroll = true;  
    }  
}  
if (startScroll) {  
    // 可以滑动。设置状态  
  
setScrollState(SCROLL_STATE_DRAGGING);  
}  
}  
  
if (mScrollState == SCROLL_STATE_DRAGGING) {  
    mReusableIntPair[0] = 0;  
    mReusableIntPair[1] = 0;  
    if (dispatchNestedPreScroll( // 处理  
nested 联动  
        canScrollHorizontally ? dx : 0,  
        canScrollVertically ? dy : 0,  
        mReusableIntPair, mScrollOffset,  
        TYPE_TOUCH
```

```
        })
        dx -= mReusableIntPair[0];
        dy -= mReusableIntPair[1];
        // Updated the nested offsets
        mNestedOffsets[0] +=
mScrollOffset[0];
        mNestedOffsets[1] +=
mScrollOffset[1];
        // scroll has initiated, prevent
parents from intercepting

getParent().requestDisallowInterceptTouchEvent(true);
    }

    mLastTouchX = x - mScrollOffset[0];
    mLastTouchY = y - mScrollOffset[1];

    if (scrollByInternal( // 调用
scrollByInternal() 让每个view自己滑动
        canScrollHorizontally ? dx : 0,
        canScrollVertically ? dy : 0,
        e)) {

getParent().requestDisallowInterceptTouchEvent(true);
    }
}
} break;

case MotionEvent.ACTION_UP: {
    mVelocityTracker.addMovement(vtev);
    eventAddedToVelocityTracker = true;
    mVelocityTracker.computeCurrentVelocity(1000,
mMaxFlingVelocity);
    final float xvel = canScrollHorizontally
        ? -
mVelocityTracker.getXVelocity(mScrollPointerId) : 0;
    final float yvel = canScrollVertically
```

```

? -
mVelocityTracker.getVelocity(mScrollPointerId) : 0;
    // 跟踪滑动速度，是否采取 fling 操作
    if (!((xvel != 0 || yvel != 0) && fling((int)xvel, (int)yvel))) {
        setScrollState(SCROLL_STATE_IDLE);
    }
    resetScroll();
} break;
}
}

```

小结

这里分两种情况：

- 没有触发 fling 操作，直接调用 scrollByInternal() 方法，最终调用 View 的 offsetLeftAndRight(offset) 方法。
- 触发 fling 操作，由 mViewFlinger.fling(velocityX, velocityY) 处理，最终在 mViewFlinger 的 run() 方法中，调用 View 的 offsetLeftAndRight(offset) 方法。

总结

看到这里不知道大家有没有被绕晕，其实不管方法是哪个，都是调用每个子 View 的 offsetLeftAndRight(offset) 来实现列表的滑动

| | <i>scrollTo</i> | <i>scrollBy</i> | <i>scrollToPosition</i> | <i>smoothScrollBy</i> | <i>smoothScrollToPosition</i> |
|----------------------|-----------------|-----------------|-------------------------|-----------------------|-------------------------------|
| onScrolled | 无反应 | 触发 | 位置可见无反应，否则触发 | 触发 | 位置可见无反应，否则触发 |
| onScrollStateChanged | 无反应 | 无反应 | 无反应 | 触发 | 位置可见无反应，否则触发 |

手势滑动两者都会触发，只不过因为放不下了，没有在表格里展示 :)

第三节 事件分发与嵌套滚动

3.1 一篇文章让你轻松弄懂NestedScrollingParent & NestedScrollingChild

虽然很早之前使用CoordinatorLayout时就认识过nestedScrollingChild和nestedScrollingParent，也看多很多博客，但每次看着就不知所云了，所以这篇文章，我们就以问题为线索，带着问题找答案。

3.1.1 谁实现 NestedScrollingChild，谁实现NestedScrollingParent？

在实际项目中，我们往往会遇到这样一种需求，当viewA还显示的时候，往上滑动到viewA不可见时，才开始滑动viewB，又或者向下滑动到viewB不能滑动时，才开始向上滑动viewC。如果列表滑动、上拉加载和下拉刷新的view都封装成一个组件的话，那滑动逻辑就是刚刚这样。而这其中列表就要实现nestedScrollingChild，最外层的Container实现nestedScrollingParent。如果最外层的container希望在其它布局中仍然能够将滑动事件继续往上冒泡，那么container在实现nestedScrollingParent的同时也要实现nestedScrollingChild。如下示意图所示。

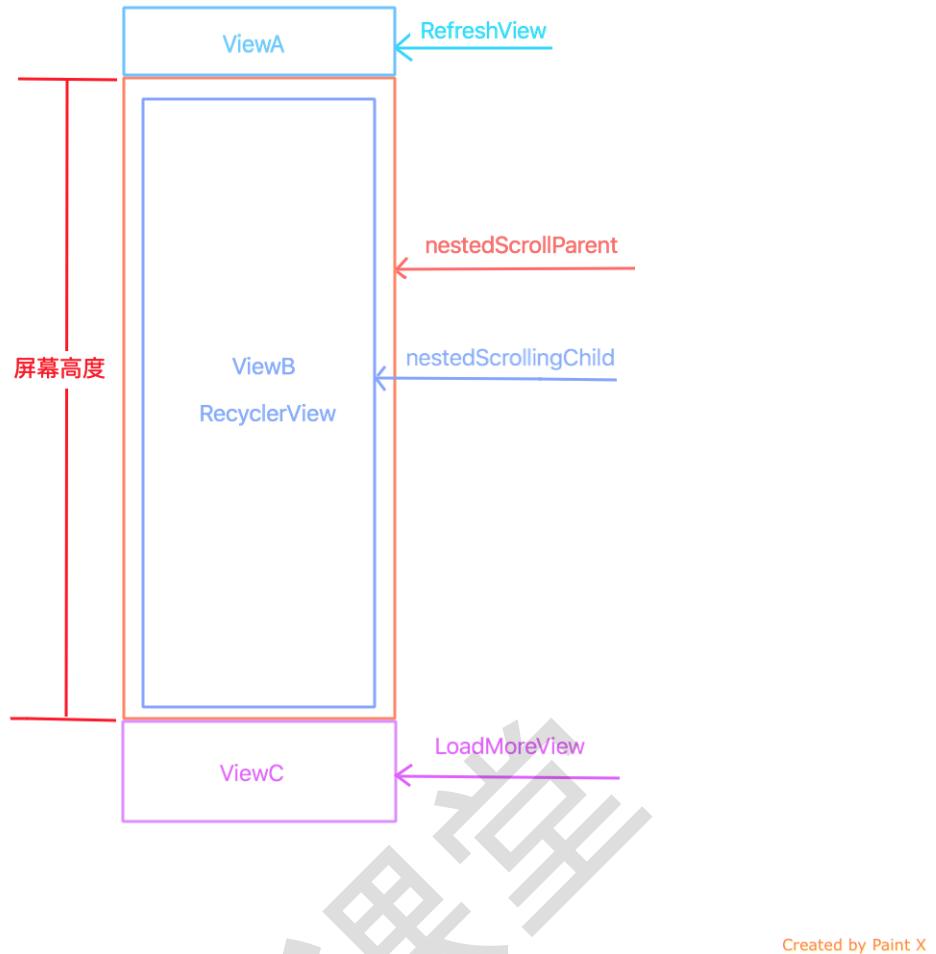


示意图.png

所以这个问题的答案：

触发滑动的组件或者接受到滑动事件且需要继续往上传递的是
`nestedScrollingChild`.

是`nestedscrollingchild`的父布局，且需要消费传递的滑动事件就是
`nestedscrollingparent`.

我们今天的最后也会给出如何利用`nestedscrollingchild`和
`nestedscrollingparent`来自定义一个集上拉加载和下拉刷新的组件。

3.1.2 滑动事件如何在二者之间传递和消费的？

3.1.2.1 你能一眼认出这是`child`还是`parent`的api吗？

首先呢，我们要看一下`nestedscrollingchild`和
`nestedscrollingparent`有哪些api.

```
public interface Nestedscrollingchild {
```

```
    public void setNestedScrollingEnabled(boolean enabled);

    public boolean isNestedScrollingEnabled();

    public boolean startNestedScroll(int axes);

    public void stopNestedScroll();

    public boolean hasNestedScrollingParent();

    public boolean dispatchNestedScroll(int dxConsumed,
int dyConsumed, int dxUnconsumed, int dyUnconsumed, int[]
offsetInwindow);

    public boolean dispatchNestedPreScroll(int dx, int dy, int[] consumed, int[] offsetInwindow);

    public boolean dispatchNestedFling(float velocityX, float velocityY, boolean consumed);

    public boolean dispatchNestedPreFling(float velocityX, float velocityY);
}
```

```
public interface NestedscrollingParent {

    public boolean onStartNestedScroll(view child, view target, int nestedScrollAxes);

    public void onNestedScrollAccepted(view child, view target, int nestedScrollAxes);

    public void onStopNestedScroll(view target);
```

```
    public void onNestedScroll(View target, int dxConsumed, int dyConsumed, int dxUnconsumed, int dyUnconsumed);  
  
    public void onNestedPreScroll(View target, int dx, int dy, int[] consumed);  
  
    public boolean onNestedFling(View target, float velocityX, float velocityY, boolean consumed);  
  
    public int getNestedScrollAxes();  
}
```

这里呢，我删掉了注释，我不想翻译那些注释放在上面的代码中，这样你就会将注意力放在我的注释中，然后陷入了咬文嚼字，最后感叹为什么每个字我都认识，连在了一起我怎么就看不懂的自我否定中。之所以把上面的代码放出来，是为了我后面简述时，你不用一边看文章，一边还要切过去看源码看这个 api 是属于 child 还是属于 parent 的。

其实这里给你一个分辨是 child 和 parent 的 api 的一个小诀窍，因为 child 是产生滑动的造势者，所以它的 api 都是以直接的动词开头，而 parent 的滑动响应是 child 通知 parent 的，所以都是以监听 on 开头，这样就记住了。

parent ----> onxxxx()

child -----> verbxxxx()

嗯，废话了好像很多了，这里我们要回到问题上，滑动事件如何在 child 和 parent 之间传递和消费掉的呢？

3.1.2.2 滑动事件的是如何传递的

那既然能传递，说明这个滑动事件一定产生了，如何产生滑动事件？当然是用户手指在屏幕上滑动了呀。为了不说的这么枯燥，我们拿最熟悉熟悉的小伙伴 RecyclerView 来作为 nestedScrollingChild 讲解。这里我引入的版本是：25.3.1

```
implementation 'com.android.support:recyclerview-v7:25.3.1'
```

2.2.1 滑动事件传递从哪里产生?

```
switch (action) {  
    case MotionEvent.ACTION_DOWN:  
        int nestedScrollAxis =  
ViewCompat.SCROLL_AXIS_NONE;  
        if (canScrollHorizontally) {  
            nestedScrollAxis |=  
ViewCompat.SCROLL_AXIS_HORIZONTAL;  
        }  
        if (canScrollVertically) {  
            nestedScrollAxis |=  
ViewCompat.SCROLL_AXIS_VERTICAL;  
        }  
        startNestedScroll(nestedScrollAxis);  
    }  
    break;  
}
```

这里，我们可以发现，当我的小手按在 RecyclerView 上时，调用了 nestedScrollingChild 的 startNestedScroll(nestedScrollAxis)，这里我们再多让我们的大脑接受一点信息，那就是这个方法的参数：nestedScrollAxis，滑动的坐标轴。RecyclerView 是不是既可以水平滑动，又可以纵向滑动，那这里就是传递的就是 RecyclerView 可以滑动的坐标轴。

发现了 startNestedScroll(axis)，看看走到了哪里。

```
@Override  
public boolean startNestedScroll(int axes) {  
    return  
getScrollingChildHelper().startNestedScroll(axes);  
}
```

这里，我们发现又出来一个类：`NestedscrollingChildHelper`. 里面好像又有一些滑动的api。读到这里，不要怕，心态要稳住，不要崩塌了。给自己吃颗定心丸，我能行。

我们先看一眼，这个`childHelper`的这个方法干了啥？

```
public boolean startNestedScroll(int axes) {  
    if (hasNestedscrollingParent()) {  
        // Already in progress  
        return true;  
    }  
    if (isNestedscrollingEnabled()) {  
        ViewParent p = mView.getParent();  
        View child = mView;  
        while (p != null) {  
            if  
(ViewParentCompat.onStartNestedScroll(p, child, mView,  
axes)) {  
                mNestedscrollingParent = p;  
  
ViewParentCompat.onNestedScrollAccepted(p, child, mView,  
axes);  
                return true;  
            }  
            if (p instanceof View) {  
                child = (View) p;  
            }  
            p = p.getParent();  
        }  
    }  
    return false;  
}
```

所有的绝妙之处就在这个方法中，这个方法我是原封不动拷贝下来，听我和你一句一句讲解。

第一句：判断 `mNestedScrollingParent` 是不是 `null`。在 `NestedScrollingChildHelper` 这个类，全类只有两处给它赋值了，一个赋有值，就是上面代码中的 `while` 循环里面，一个是赋空值，在方法 `stopNestedScroll`，这个方法什么时候调用啊，在你美丽的小手离开屏幕的时候。所以只要你的小手在屏幕上，这个 `startNestedScroll` 这个方法只会调用一次。也就是通知 `parent` 我美丽的小手指要滑动啦，通知过你，我就不通知了，哪个小仙女不是傲娇的。

第二句：判断 `mIsNestedScrollingEnabled` 是否要 `true`。这个变量也是至关重要的，它的作用是要不要向上冒泡滑动事件，所以说哪天小仙女不开心了，直接调用了：`setNestedScrollingEnabled(false)`，父布局是怎么都不知道小手指有没有滑动的。

第三句 + 第四句：这里的 `p` 就是父布局了，这里的 `mView` 是在初始化这个类的时候，传递过来的，所以在 `RecyclerView` 中，可以找到这句话：

```
mScrollingChildHelper = new  
NestedScrollingChildHelper(this);
```

 这里的 `mView` 就是 `RecyclerView` 这位小仙女啦。

第五句：进入 `while` 循环了，为什么这里要 `while` 循环，因为它要确保使命必达，不管我的父布局有多深，我都要找到你，并通知到你。

第六句：`if` 里的逻辑说明，如果 `parent` 监听到即将要在这个轴上有滑动事件，并且正是 `parent` 需要的事件，那么就会调用 `onNestedScrollAccept`。这里的 `viewParentCompat.onStartNestedScroll(p, child, mView, axes)` 会最终调用到实现 `nestedScrollingParent` 组件中的 `onStartNestedScroll` 方法，这个方法就是 `parent` 判断收到该滑动通知时，是不是天时地利人和，如果是，我就返回 `true`，后面一系列的小手指滑动都要告知我。如果返回 `false`，说明 `parent` 此时在处理别的事情，后面小手指滑动的弧线再怎么优美，都不要来烦我。

第七句：`onNestedScrollAccepted` 说明 parent 正式接收了此 child 也就是 recyclerView 的滑动通知，最终会调用到 parent 的 `onNestedScrollAccept` 方法中，如果此 parent 还实现了接口 `nestedScrollingChild`，可以在这个方法继续向 parent 的 parent 上报了。

所以整个流程可以概括为：通知

```
ACTION_DOWN  
--> child.startNestedScroll  
--> childHelper.startNestedScroll  
--> parent.onStartNestedScroll  
--> parent.onNestedScrollAccept
```

2.2.2 小手指滑动的时候，child 和 parent 之间是如何通信的？

```
case MotionEvent.ACTION_MOVE: {  
    if (dispatchNestedPreScroll(dx, dy,  
        mScrollConsumed, mScrollOffset)) {  
        dx -= mScrollConsumed[0];  
        dy -= mScrollConsumed[1];  
        vtev.offsetLocation(mScrollOffset[0],  
            mScrollOffset[1]);  
    }  
  
    if (mScrollState == SCROLL_STATE_DRAGGING) {  
        mLstTouchX = x - mScrollOffset[0];  
        mLstTouchY = y - mScrollOffset[1];  
  
        if (scrollByInternal(canScrollHorizontally ?  
            dx : 0, canScrollVertically ? dy : 0,  
            vtev)) {  
  
            getParent().requestDisallowInterceptTouchEvent(true);  
        }  
    }  
}  
break;  
}
```

这里呢，有两个重要的方法：`dispatchNestedPreScroll` 和 `scrollByInternal`。

第一句：`dispatchNestedPreScroll(dx, dy, consumed, offsetInwindow)`, 这里的参数中只有 `dx`, `dy` 两个参数在前面赋值了，而后面两个参数在哪里操作的呢？这里我们留个问号？首先这个方法会走到 `NestedScrollingChildHelper` 类中的方法：

`dispatchNestedPreScroll` 调用

```
viewParentCompat.onNestedPreScroll(mNestedScrollingParent,  
mView, dx, dy, consumed);
```

```
public boolean dispatchNestedPreScroll(int dx, int dy,  
int[] consumed, int[] offsetInwindow) {  
  
    viewParentCompat.onNestedPreScroll(mNestedScrollingParent  
, mView, dx, dy, consumed);  
}
```

最终目的地来到了 `parent` 的 `onNestedPreScroll()`。所以我们可以大胆猜测，`consumed`, `offsetInwindow`, 是在 `parent` 这里赋值的，当然你可以不用赋值，不赋值的话，值也就是保留上一次的值。

```
dx -= mScrollConsumed[0];  
dy -= mScrollConsumed[1];
```

`dispatchNestedPreScroll()` 这个方法返回 `true` 后，发现重新计算了 `dx`, `dy`, 在方法 `scrollByInternal()` 方法中，用的是最新的 `dx`, `dy` 值。说明当小手指产生滑动位移的时候，先分发给 `parent`，让 `parent` 先消耗，并在方法中将 `parent` 消耗的位移传递过来，那么剩下的位移，ok，那充当 `child` 的 `Recyclerview` 内部消费了。

```
    if (scrollByInternal(canScrollHorizontally ? dx : 0, canScrollVertically ? dy : 0, vtev)) {  
  
        getParent().requestDisallowInterceptTouchEvent(true);  
    }  

```

```
boolean scrollByInternal(int x, int y, MotionEvent ev) {  
    int unconsumedX = 0, unconsumedY = 0;  
    int consumedX = 0, consumedY = 0;  
    if (mAdapter != null) {  
        if (y != 0) {  
            consumedY = mLayout.scrollVerticallyBy(y, mRecycler, mState);  
            unconsumedY = y - consumedY;  
        }  
        if (dispatchNestedScroll(consumedX, consumedY, unconsumedX, unconsumedY, mScrollOffset)) {  
            // Update the last touch co-ords, taking any  
            scroll offset into account  
            mLasteTouchX -= mScrollOffset[0];  
            mLasteTouchY -= mScrollOffset[1];  
            if (ev != null) {  
                ev.offsetLocation(mScrollOffset[0],  
mScrollOffset[1]);  
            }  
            mNestedOffsets[0] += mScrollOffset[0];  
            mNestedOffsets[1] += mScrollOffset[1];  
        }  
        return consumedX != 0 || consumedY != 0;  
    }  

```

第二句：`scrollByInternal()`就是内部滑动消耗了，在这个方法里面，我们发现继续往`parent`分发了事件：

`dispatchNestedScroll(consumeX, consumeY, unconsumeX, unconsumeY)`，把自己未消耗的滑动位移继续移交给`parent`，这个时候最终会走到`parent`的方法：`onNestedScroll()`。在这里，如果`parent`还实现了`nestedScrollingChild`，可以将未消耗的滑动位移继续移交给自己的`parent`。

```
@Override  
    public void onNestedScroll(View target, int dxConsumed,  
        int dyConsumed, int dxUnconsumed, int dyUnconsumed) {  
        if(isNestedScrollingEnabled()) {  
            dispatchNestedScroll(dxConsumed, dyConsumed,  
                dxUnconsumed, dyUnconsumed, mParentOffsetInWindow);  
        }  
    }
```

所以我们可以总结如下：通信

```
ACTION_MOVE : 小手指滑动位移为: dy  
--> childHelper.dispatchNestedPreScroll(dy)  
--> parent.onNestedPreScroll(dy), consumedY =  
parent.onNestedPreScroll(dy)  
--> dy' = dy - consumeY recyclerView.scrollByInternal(dy')  
unconsumeY = dy' - recyclerView.scrollByInternal(dy')  
--> parent.startNestedScroll(unconsumeY)
```

2.2.3 小手指滑累了，离开屏幕时，又有哪些事件传递？

```
case MotionEvent.ACTION_UP: {
    if (!((xvel != 0 || yvel != 0) && fling((int)xvel, (int)yvel))) {
        setScrollState(SCROLL_STATE_IDLE);
    }
    resetTouch();
}
break;
```

```
public boolean fling(int velocityX, int velocityY) {
    if (!dispatchNestedPreFling(velocityX,
velocityY)) {
        final boolean canScroll = canScrollHorizontal
|| canScrollVertical;
        dispatchNestedFling(velocityX, velocityY,
canScroll);
        if (canScroll) {
            mViewFlinger.fling(velocityX, velocityY);
            return true;
        }
    }
    return false;
}
```

```
private void resetTouch() {
    stopNestedScroll();
}
```

这里我们发现先是 child 执行 fling 方法，也就是当手松开时仍然有速度，那么会执行一段惯性滑动，而在这惯性滑动中，这里就很奇妙了，先是通过 dispatchNestedPreFling() 将滑动速度传递给 parent，如果 parent 不消耗的话，再次通过 dispatchNestedFling 向 parent 传递，只是这次的传递会带上 child 自己是否有能力消费惯性滑动，最后不管

`parent`有没有消费，`child`也就是`recyclerview`都会执行自己的`fling`.也就是：

```
mViewFlinger.fling(velocityX, velocityY);
```

走完了惯性滑动，就会走到`stopNestedScroll()`. 按照上面的逻辑处理，我们应该可以猜到接下来的逻辑就是走到`NestedScrollingChildHelper`这个类。然后目的地会到达`parent`的`onStopNestedScroll`方法。这里，`parent`就可以处理当小手指离开屏幕时的一些逻辑了。这条路很简单，没有返回值，也没有传递什么变量。还是很好理解的。

```
public void stopNestedScroll() {  
    if (mNestedScrollingParent != null) {  
  
        ViewParentCompat.onStopNestedScroll(mNestedScrollingParent,  
            mview);  
        mNestedScrollingParent = null;  
    }  
}
```

这里呢，我们可以总结如下：收尾

`ACTION_UP`

```
--> childHelper.dispatchNestedPreFling  
--> parent.onNestedPreFling  
--> childHelper.dispatchNestedFling  
--> parent.onNestedFling  
--> child.fling  
--> childHelper.stopNestedScroll  
--> parent.onStopNestedScroll
```

这样，我们整个`nestedScrollingChild`和`nestedScrollingParent`之间的丝丝缕缕都讲解完了。

3.1.2.3实践

这里我们利用nestedscrollingChild和nestedscrollingParent实现的自定义上拉加载，下拉刷新的控件。

<https://github.com/thh0613/nestedScrollDemo>

3.2 动态化页面构建方案

电商类的APP使用居多

3.2.1 Android | Tangram动态页面之路（二）介绍

本文主要对Tangram和vlayout的一些概念进行介绍。

vlayout

因为Tangram底层基于vlayout，所以需要先了解下vlayout。

首先，在view上的性能消耗通常有以下几种：

- 布局嵌套导致多重measure/layout

可以使用ConstraintLayout或RelativeLayout减少布局嵌套

- view的频繁创建与销毁

列表使用RecyclerView来复用布局

- xml转换成view解析过程产生的内存和耗时

如果列表的样式不多，使用RecyclerView的复用机制可以避免大量的xml解析；如果样式比较多比如商品图墙等，则有必要把xml解析提前到编译期，在编译期根据注解将xml转成对应的view类，直接使用view类创建ViewHolder，当然这么做会势必会增大包体积，需要克制使用

然后，vlayout主要解决前两点，做到复杂布局下扁平和细粒度复用。

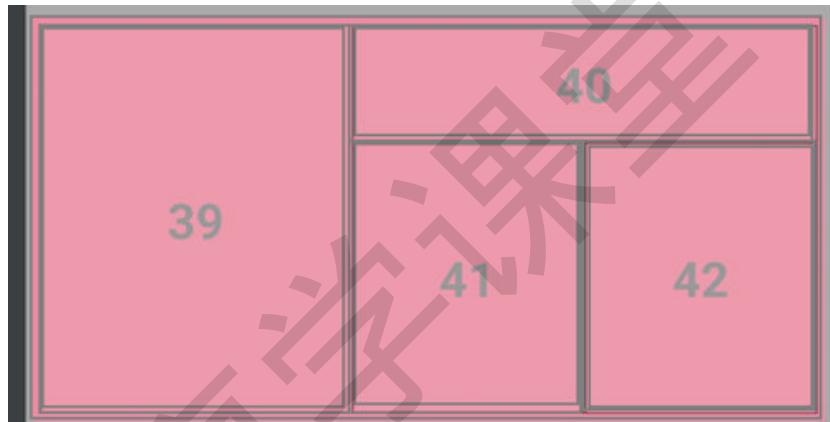
常规的RecyclerView使用：

```
// 设置适配器，管理数据源和view  
recyclerView.setAdapter()  
// 设置LayoutManager，指定布局方式  
recyclerView.setLayoutManager()
```

其中 `LayoutManager` 有3种，

```
LinearLayoutManager extends LayoutManager //线性  
GridLayoutManager extends LinearLayoutManager //网格  
StaggeredGridLayoutManager extends LayoutManager //瀑布流
```

在面对比较复杂的布局时，如1拖3样式，

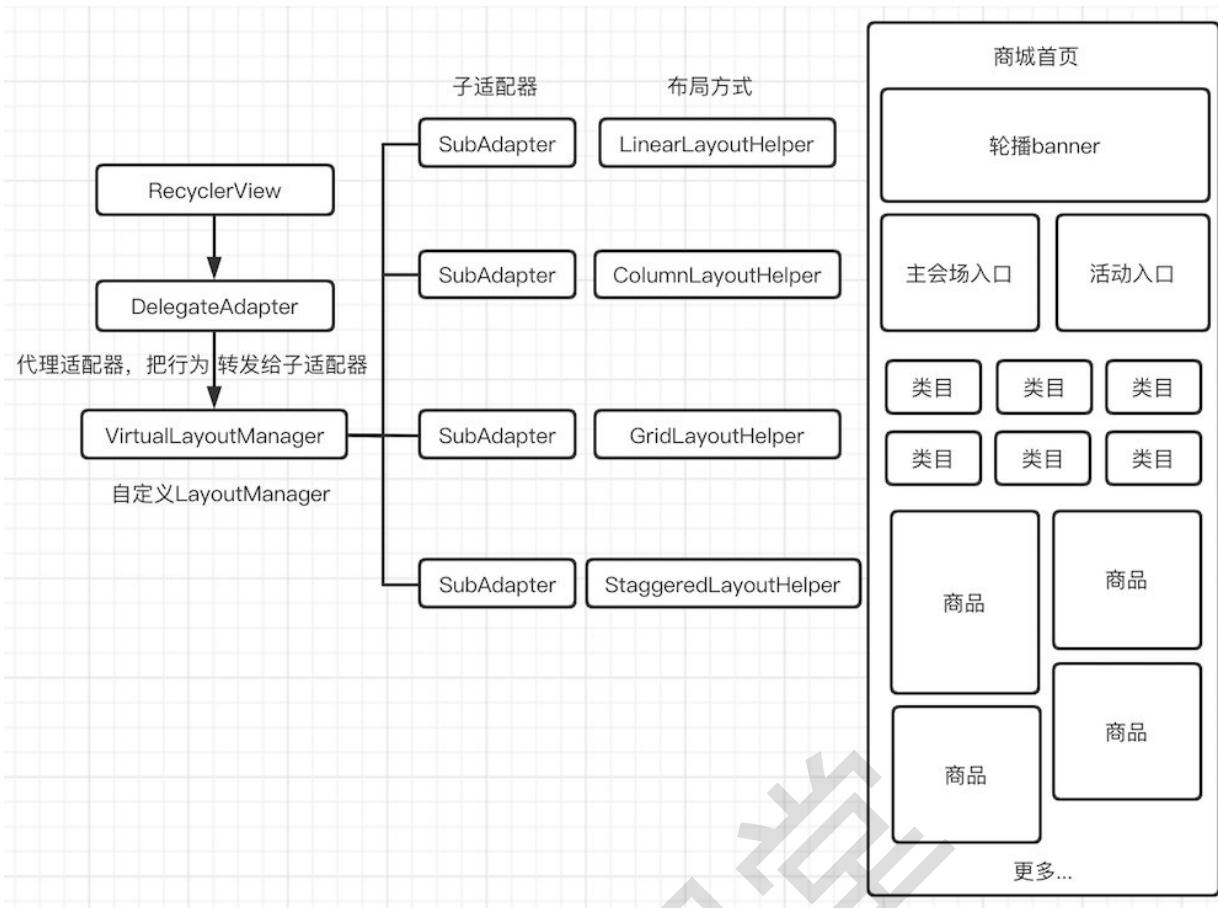


通常只能在1拖3外边套上一层layout，然后使用 `LinearLayoutManager` 实现。为了解决这个问题，

vlayout自定义了一个VirtualLayoutManager，它继承自 `LinearLayoutManager`；引入了 `LayoutHelper` 的概念，它负责具体的布局逻辑；VirtualLayoutManager管理了一系列LayoutHelper，将具体的布局能力交给LayoutHelper来完成，每一种LayoutHelper提供一种布局方式，框架内置提供了几种常用的布局类型，包括：网格布局、线性布局、瀑布流布局、悬浮布局、吸边布局等。这样实现了混合布局的能力，并且支持扩展外部，注册新的LayoutHelper，实现特殊的布局方式。

引用自[苹果核 - Tangram 的基础 —— vlayout \(Android\)](#)

大致意思是这样，



在`vlayoutDemo`中，使用代码是这样，

```
//VLayoutActivity.java

//子适配器集合
List<DelegateAdapter.Adapter> adapters = new LinkedList<>();
//创建子适配器，需指定其布局方式
SubAdapter subAdapter1 = new SubAdapter(new
    LinearLayoutHelper());
adapters.add(subAdapter1);
SubAdapter subAdapter2 = new SubAdapter(new
    ColumnLayoutHelper());
adapters.add(subAdapter2);

//把子适配器集合设置给代理适配器
delegateAdapter.setAdapters(adapters);

recyclerView.setAdapter(delegateAdapter);
```

可以看到，随着布局样式越来越多，代码量也会越来越多，于是，用json模板描述页面的Tangram诞生了。

Tangram

把vlayout直接给到业务方使用，这样的接入成本是不能接受的，于是需要屏蔽掉vlayout细节，让业务方用的更舒服。至于为何要引入json模板，[需求背景](#)里已经做过介绍。

Tangram的意思是七巧板，旨在用七巧板的方式拼凑出各式各样的页面。他抽象了两个概念，Card和cell，Card用于描述布局方式，cell用于描述在这个布局方式下，用什么样的view去展示，比如Tangram Demo里的data.json，

```
[  
  {  
    "type": "container-oneColumn", //Card, 布局方式  
    "items": [  
      {  
        "imgUrl":  
          "https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-  
          110-72.png",  
        "arrowImgUrl":  
          "https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-  
          110-72.png",  
        "title": "标题1",  
        "type": "vvtest" //Cell, 具体展示的view  
      },  
      {  
        "imgUrl":  
          "https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-  
          110-72.png",  
        "arrowImgUrl":  
          "https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-  
          110-72.png",  
        "title": "标题2",  
        "type": "vvtest"  
      }  
    ]  
]
```

```
        }  
    ]  
}  
]
```

这些布局方式 card，在 Tangram 内部会进行注册，

```
//TangramBuilder.java  
void installDefaultRegistry(final DefaultResolverRegistry  
registry) {  
    // built-in cards  
    registry.registerCard(TYPE_CONTAINER_BANNER,  
BannerCard.class);  
    registry.registerCard(TYPE_SINGLE_COLUMN_COMPACT,  
singleColumnCard.class);  
    //...  
}
```

布局方式确定好后，需要具体的 View 来展示，也就是 cell，比如单图 singleImageView，纯文本 RatioTextView 等等，这些 cell 则需手动注册，如果是偏业务的 cell，可以在业务层按需注册，如果是更抽象的通用 cell，则应该下沉到基础库里全局注册，更抽象的 cell 意味着需要提供更为通用的配置属性，能提供给更多不同的业务方使用。cell 的手动注册如下，

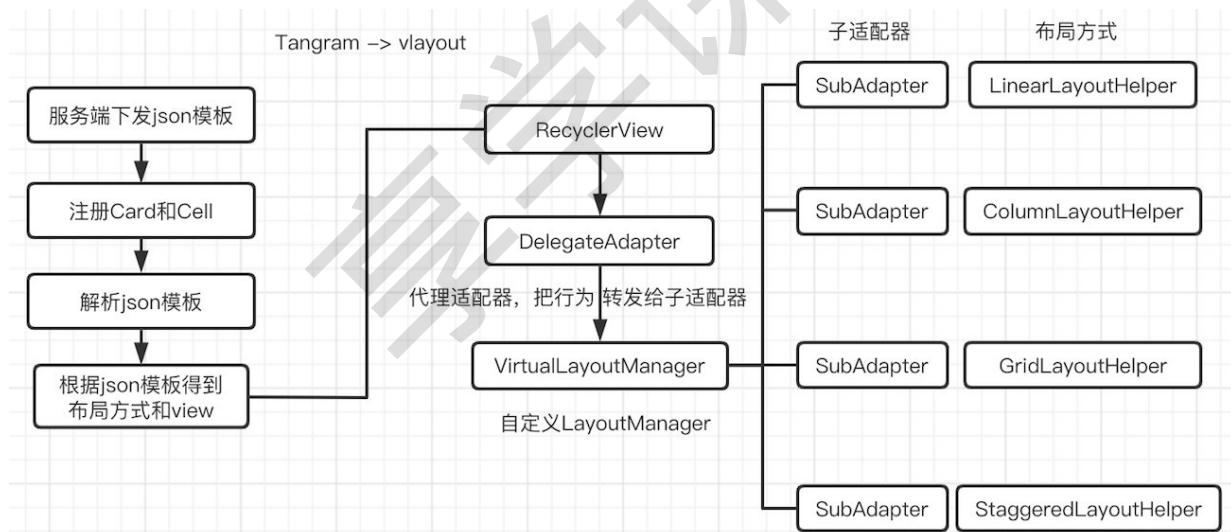
```
//TangramActivity.java  
void onCreate(Bundle savedInstanceState) {  
    //Step 3: register business cells and cards  
    builder.registerCell(1, TestView.class);  
    builder.registerCell(2, SimpleImgView.class);  
  
    //绑定 recyclerView  
    engine.bindView(recyclerView);  
  
    JSONArray data = new JSONArray(new  
String(getAssertsFile(this, "data.json")));
```

```
//设置json数据  
engine.setData(data);  
}
```

Card和Cell都注册好了，通过TangramEngine将数据设置进去，跟进去可以看到，Tangram把布局能力交给了vlayout，

```
//GroupBasicAdapter extends VirtualLayoutAdapter  
void setData(List<L> cards, boolean silence) {  
    //把Card转成LayoutHelper  
    setLayoutHelpers(transformCards(cards, mData,  
mCards));  
}
```

Tangram把json模板中描述的一个个card解析成了所对应的vlayout的布局方式LayoutHelper，



值得注意的是，Tangram Demo里的json模板都是包含了业务数据的，这么做可能是为了剔除掉网络请求的代码，方便开源学习，而在实际业务中不太可能把数据绑定在模板里，这样模板会很臃肿，我们要做的是，用模板描述页面结构和数据源，而非数据本身，如，

```
{  
    "template": [  
        {
```

```
        "card": "container-banner", //轮播图模块，使用
banner布局方式
        "cell": "imageView", //具体的view就是一张图片
        "data": "makeup:banner" //数据来源于聚合接口，key
为banner
    },
{
    "card": "container-fiveColumn", //类目模块，使
用column布局方式
    "cell": "imageAndTextView", //具体的view是上图下
文本
    "data": "makeup:category" //数据来源于聚合接口，
key为category
},
{
    "card": "container-waterfall", //商品流模块，使
用Staggered布局方式
    "cell": "goodsView", //具体的view是商品样式
    "data": "request:recommend" //数据来源于
request, key为recommend
}
]
```

既然json模板可以由后端下发，也就意味着，我们可以让运营同学通过后台拖动模块搭建页面，选择资源位设置数据源，然后生成json模板下发，开发同学从此就可以解放双手，做~更有趣~（更有挑战）的事情了。

3.2.2 Android动态界面开发框架Tangram使用完整教程

熟悉阿里出品的vlayout的读者都知道，vlayout极大地扩展了RecyclerView的LayoutManager，从而为RecyclerView提供了一组布。使用该开源项目，可以让我们在同一个RecyclerView里处理线性、网格等等各种复杂的布局。然而，使用vlayout时，一切都需要用Java代码实现，而且都是写在客户端上，一旦需要修改就必须发版，不是很灵活，

于是阿里又提出了Tangram，其可以使用json来配置布局，可以说极大地提升了灵活性。下面我们来学习一下如何使用Tangram。

1 Tangram的概念

Tangram 是阿里出品的用于快速实现组合布局的框架模型，在手机天猫 Android 及 iOS 版广泛使用。



中文翻译为七巧板，即该框架提供一系列基本单元布局，通过快速拼装就能搭建出一个具备多种布局的页面。

Tangram提供了流式布局、滚动布局，瀑布流布局，固定布局等数种布局样式，布局提供样式参数供调整，布局内部也可填充任意的视图(View)，使Native开发的页面具备一定的动态性，并提供极致的性能。

Tangram包含的特点如下：

- Android iOS 双平台支持，iOS 版本参考开源库 [Tangram-iOS](#)。

- 通过 json 创建页面视图，并提供了默认的解析器。
- 可轻松实现页面视图的回收与复用。
- 框架提供多种默认的布局方式。
- 通过 json 数据或代码支持自定义布局样式。
- 高性能，基于vlayout。
- 支持扩展功能模块。

下面来看看如何使用Tangram。

2 Tangram使用步骤

2.1 引入依赖

在APP的**build.gradle**中添加：

```
implementation 'com.alibaba.android:tangram:3.3.6@aar'  
// we added rxjava in latest version, so need compile  
rxjava  
implementation 'io.reactivex.rxjava2:rxjava:2.1.12'  
implementation 'io.reactivex.rxjava2:rxandroid:2.0.2'
```

其中，Tangram的版本号可以改成最新的，最新版本号可以在这个链接找到：<https://github.com/alibaba/Tangram-Android/releases>

另外还要添加**VirtualView**和**UltraViewPager**这两个库，否则运行时会出现问题：

```
implementation  
('com.alibaba.android:virtualview:1.4.6@aar') {  
    transitive = true  
}  
implementation  
('com.alibaba.android:ultraviewpager:1.0.7.7@aar') {  
    transitive = true  
}
```

VirtualView的最新版本号可以在这里找到：

<https://github.com/alibaba/Virtualview-Android/releases>

2.2 初始化 Tangram 环境

应用全局只需要初始化一次，需要提供一个图片加载器，例如使用Glide库或Picasso库，代码如下：

```
TangramBuilder.init(context, new IInnerImageSetter() {  
    @Override  
    public <IMAGE extends ImageView> void  
    doLoadImageUrl(@NonNull IMAGE view,  
                    @Nullable String url) {  
        //假设你使用 Picasso 加载图片  
        Picasso.with(context).load(url).into(view);  
    }  
}, ImageView.class);
```

2.3 初始化 TangramBuilder

在 Activity 中初始化TangramBuilder，假如你的 Activity 是 TangramActivity，则代码如下：

```
TangramBuilder.InnerBuilder builder =  
TangramBuilder.newInnerBuilder(TangramActivity.this);
```

2.4 注册自定义的卡片和组件

注册组件的方式有如下3种：

(1) 注册绑定组件类型和自定义View，示例代码：

```
builder.registerCell("type", TestView.class);
```

(2) 注册绑定组件类型、自定义 model、自定义View，示例代码：

```
builder.registerCell("type", TestCell.class,  
TestView.class);
```

(3) 注册绑定组件类型、自定义model、自定义ViewHolder，示例代码：

```
builder.registerCell("type", TestCell.class, new  
ViewHolderCreator<>(R.layout.item_holder,  
TestViewHolder.class, TestView.class));
```

这里先不做详解，关于卡片和组件的详细使用请参见第3节。

2.5 生成 TangramEngine 实例

在上述基础上调用：

```
TangramEngine engine = builder.build();
```

2.6 绑定业务 support 类到 engine

Tangram 内部提供了一些常用的 support 类辅助业务开发，具体请见3.3 节，使用方式有如下3种：

```
engine.register(SimpleClickSupport.class, new
XXClickSupport());
engine.register(CardLoadSupport.class, new
XXCardLoadSupport());
engine.register(ExposureSupport.class, new
XXExposureSuport());
```

2.7 绑定 RecyclerView

```
engine.bindView(recyclerView);
```

2.8 监听 RecyclerView 的滚动事件

```
recyclerView.addOnScrollListener(new
RecyclerView.OnScrollListener() {
    @Override
    public void onScrolled(RecyclerView recyclerView, int
dx, int dy) {
        super.onScrolled(recyclerView, dx, dy);
        //在 scroll 事件中触发 engine 的 onScroll，内部会触发
        //需要异步加载的卡片去提前加载数据
        engine.onScrolled();
    }
});
```

2.9 设置悬浮类型布局的偏移（可选）

如果你的 RecyclerView 上方还覆盖有其他 view，比如底部的 tabbar 或者顶部的 actionbar，为了防止悬浮类 view 和这些外部 view 重叠，可以设置一个偏移量。此功能需要额外引入 vlayout

(<https://github.com/alibaba/vlayout>)。代码如下：

```
engine.getLayoutManager().setFixOffset(0, 40, 0, 0);
```

2.10 设置卡片预加载的偏移量（可选）

在页面滚动过程中会触发 engine.onScrolled() 方法，会去寻找屏幕外需要异步加载数据的卡片，默认往下寻找 5 个，让数据预加载出来，我们也可以修改这个偏移量，代码如下：

```
engine.setPreLoadNumber(3);
```

2.11 加载数据并传递给 engine

数据一般是调用接口加载远程数据，这里演示的是 mock 加载本地的数据：

```
byte[] bytes = Utils.getAssertsFile(this, "data.json");
if (bytes != null) {
    String json = new String(bytes);
    try {
        JSONArray data = new JSONArray(json);
        engine.setData(data);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

2.12 退出的时候销毁 engine

```
engine.destroy();
```

通过主动调用 destroy 方法，可以释放内部的资源，比如清理 adapter、清理事件总线缓存的未处理消息、注销广播等。注意调用 destroy 方法之后就不需要调用 unbind 方法了。

3.3 组件与布局

3.1.1 页面概念模型

我们将一个普通的列表页面结构化成树状结构：分别是页面、布局（卡片）和组件。一个页面下面可以挂载多个布局或者组件，一个布局下面可以挂载多个组件，整体是一个树状结构描述。每一层次都有各自的职责，如下图：



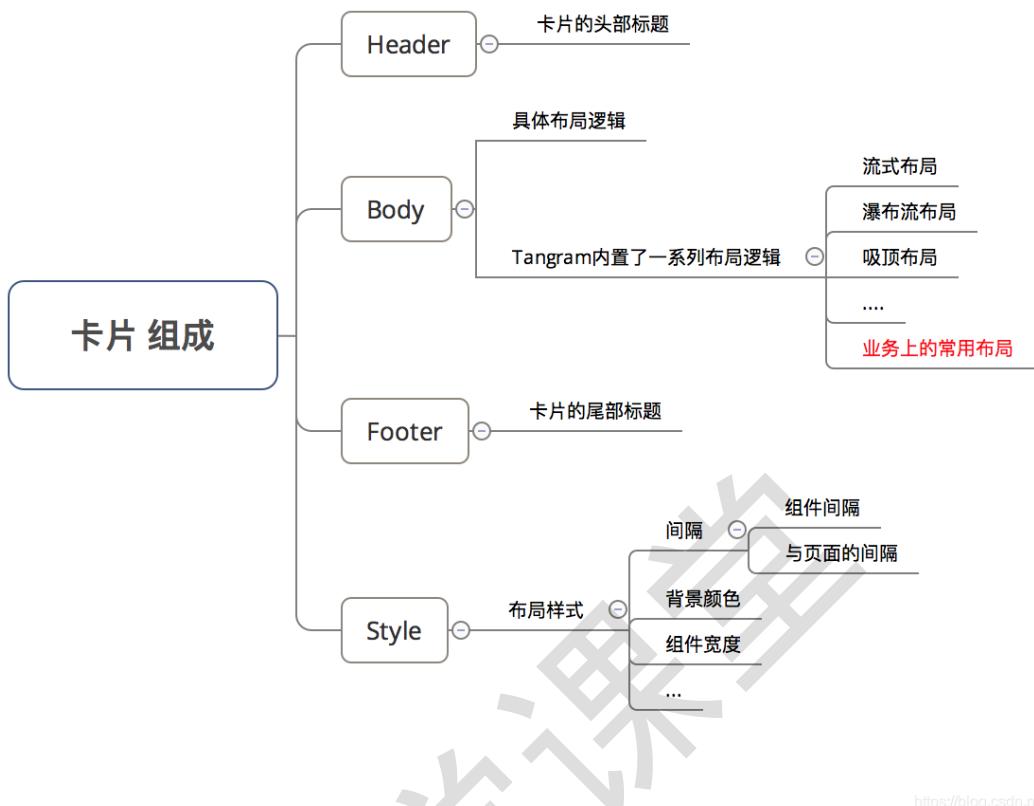
3.1.1 页面

如上图所示，一个页面包含一个卡片列表，每个卡片持有一个组件列表，整个页面是一个页面 - 布局 - 组件的树状结构。它要求整体可滚动，并且能按照组件的类型去回收复用。

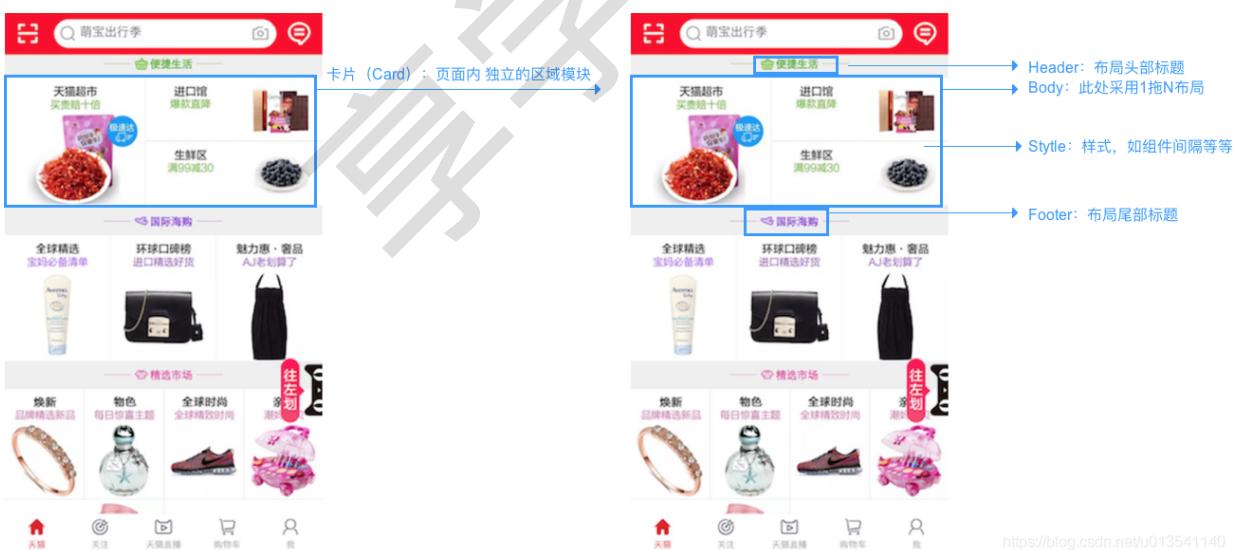
在实现上，在 Tangram-iOS 里，它是基于 LazyScrollView 的页面容器，在 Tangram-Android 里，它是基于 vlayout 构建的 RecyclerView。

3.1.2 布局（卡片）

布局的主要职责是负责对组件进行布局，它有四个组成：**header**、**footer**、**body**、**style**，如下图所示：



<https://blog.csdn.net/u013541140>



<https://blog.csdn.net/u013541140>

最重要的是 body 部分，它包含了内嵌的组件，如果布局没有 body，即没有组件，也就不在视觉上做渲染。卡片的布局也就是对 body 里包含的组件来进行布局。Tangram 内置了一系列布局能力对组件进行布局，包括流式布局、瀑布流布局、吸顶布局、悬浮布局、轮播布局等等，基本上常见的布局方式都可以覆盖到。header、footer 是卡片的标题和尾部，目前只有轮播卡片、通用流式卡片支持 header、footer。style 是对布局

样式的描述，所有布局会有一些通用的样式属性比如边距、间距，也有一些特有的比如宽高比，通过样式的描述，可以让布局能力更加丰富。

布局描述就是一种布局类型的声明，因为框架已经内置了布局能力，只需要声明采用哪一种布局方式，因此不需要布局模板。如果框架的内置布局能力满足不了需求，还可以自定义扩展新的布局类型注册到 Tangram 里。

以下是一个布局的 json 描述示例（**type**, **style**, **header**, **footer**, **items** 都是关键字）：

```
[  
  {  
    "type": "container-oneColumn", ---> 描述布局类型  
    "style": { ---> 描述样式  
      ...  
    },  
    "header": { ---> 描述header  
    },  
    "items": [ ---> 描述组件列表  
      ...  
    ],  
    "footer": { ---> 描述footer  
    }  
  },  
  ...  
]
```

3.1.3 组件

组件的职责就是负责基本的 UI 展示和交互，它是按照业务划分的最小单元，不像通用的 UI 框架那样会设计文本、按钮、线条那样的基础元素。

在 Tangram 里，组件长什么样，框架是不知道的，框架内也不内置组件，都是由我们接入的时候自行按需注册。同布局一样，组件的数据描述也需要提供与 UI 相关的模板，包含3部分：**类型**、**数据**、**样式**。类型是必须的，如果我们在 Tangram 里注册过这种类型，那么就能被框架解析处

理；数据也是必须的，它包含了业务信息；样式是可选的，组件可以按照自己的需求定义样式，在实现的时候解读样式数据。

以下是一个组件的 json 描述示例（**type**, **style**都是关键字）：

```
{  
    "type": "demo", ---> 描述组件类型  
    "style": { ---> 描述组件样式  
        "margin": [  
            10,  
            10,  
            10,  
            10  
        ],  
        "height": 100,  
        "width": 100  
    }  
    "imageUrl": "[URL]", ---> 业务数据  
    "title": "Sample"  
}
```

这样，就可以将多个组件的 json 数据放到布局的 items 里，然后将多个布局的 json 数据组合成一个 json 列表，就形成了一个页面。

下面的步骤，先使用 Java 进行组件的开发，并进行注册，然后使用 json 来描述整体的布局。

3.1.2 组件开发

组件分为两层：model 和 View。model 是对 json 数据的解析，View 就是我们自定义的 View。Tangram 里提供了通用 model 类型 BaseCell，其包含了对 json 数据的解析，还有位置等信息。开发组件有两种方式：

1. 采用通用 model，开发自定义 View；
2. 采用自定义 model 和自定义 View。

3.2.1 通用 model 开发组件

这种方式无需关心 model，主要是开发自定义 View。

自定义 View 有两种实现规范：

(1) 使用接口方式，避免了反射调用，性能上更优，步骤如下：

- 实现一个自定义View，比如XXTangramView；
- 实现接口**ITangramViewLifeCycle**，包含三个方法：

```
public void cellInitiated(BaseCell cell); // 绑定数据前调用  
public void postBindView(BaseCell cell); // 绑定数据时机  
public void postUnBindView(BaseCell cell); // 滑出屏幕，解除  
绑定
```

- 主要在上述**public void postBindView(BaseCell cell)**方法里完成组件业务逻辑。

下面实现一个示例，其中包含一个ImageView和一个TextView，代码如下：

```
public class CustomInterfaceview extends LinearLayout  
implements ITangramViewLifeCycle {  
    private ImageView mImageView;  
    private TextView mTextView;  
  
    public CustomInterfaceview(Context context) {  
        super(context);  
        init();  
    }  
  
    public CustomInterfaceview(Context context, @Nullable  
AttributeSet attrs) {  
        super(context, attrs);  
        init();  
    }  
  
    public CustomInterfaceview(Context context, @Nullable  
AttributeSet attrs,  
                                int defStyleAttr) {  
        super(context, attrs, defStyleAttr);  
        init();  
    }
```

```
}

private void init() {
    setOrientation(VERTICAL);
    setGravity(Gravity.CENTER);
    int padding = Utils.dip2px(getContext(), 10);
    setPadding(padding, padding, padding, padding);
    mImageView = new ImageView(getContext());
    addView(mImageView, LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
    mTextView = new TextView(getContext());
    mTextView.setPadding(0, padding, 0, 0);
    addView(mTextView, LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
}

@Override
public void cellInited(BaseCell cell) {
}

@Override
public void postBindView(BaseCell cell) {
    if (cell.pos % 2 == 0) {
        setBackgroundColor(0xffff0000);
        mImageView.setImageResource(R.mipmap.ic_launcher);
    } else {
        setBackgroundColor(0xff00ff00);
        mImageView.setImageResource(R.mipmap.ic_launcher_round);
    }
    mTextView.setText(String.format(Locale.CHINA,
"%s%d: %s", getClass().getSimpleName(),
cell.pos, cell.optParam("text")));
}

@Override
public void postUnBindView(BaseCell cell) {
```

```
    }  
}
```

(2) 使用注解方式，动态绑定数据：

- 实现一个自定义View，比如XXTangramView；
- 必须添加下面三个方法，以@CellRender注解，功能同上，只是被反向调用：

```
public void cellInitiated(BaseCell cell);  
public void postBindView(BaseCell cell);  
public void postUnBindView(BaseCell cell);
```

- 还可以为组件的每个属性实现单独的设置方法，而不是在postBindView方法里一次性绑定数据，这些方法必须以@CellRender注解，框架会在**public void postBindView(BaseCell cell)**方法调用之前调用这些数据绑定方法，示例代码：

```
@CellRender(key = "pos") //这里的key=pos表示让框架取原始json  
数据里pos字段的值传给该方法，原始数据里没有该字段，参数值会是该类型的默认值
```

```
public void setPosition(int pos) {//这里pos的类型要注意，是  
框架会以该方法声明的类型来取获取原始数据  
    textView.setText(cell.id + " pos: " + pos + " " +  
    cell.parent + " " + cell.optParam("msg"));  
    if (pos > 57) {  
        textView.setBackgroundColor(0x66cccf00 + (pos -  
50) * 128);  
    } else if (pos % 2 == 0) {  
        textView.setBackgroundColor(0xaaaff55);  
    } else {  
        textView.setBackgroundColor(0xccfafafa);  
    }  
}
```

下面采用注解的方式实现之前用接口方式实现的自定义View，代码如下：

```
public class CustomAnnotationView extends LinearLayout {  
    private ImageView mImageview;
```

```
private TextView mTextView;

public CustomAnnotationView(Context context) {
    super(context);
    init();
}

public CustomAnnotationView(Context context,
@Nullable AttributeSet attrs) {
    super(context, attrs);
    init();
}

public CustomAnnotationView(Context context,
@Nullable AttributeSet attrs,
int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init();
}

private void init() {
    setOrientation(VERTICAL);
    setGravity(Gravity.CENTER);
    int padding = Utils.dip2px(getContext(), 10);
    setPadding(padding, padding, padding, padding);
    mImageView = new ImageView(getContext());
    addView(mImageView, LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
    mTextView = new TextView(getContext());
    mTextView.setPadding(0, padding, 0, 0);
    addView(mTextView, LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
}

@Override
public void cellInited(BaseCell cell) {
```

```
@CellRender
public void postBindView(BaseCell cell) {
    if (cell.pos % 2 == 0) {
        setBackgroundColor(0xff0000ff);

        mImageView.setImageResource(R.mipmap.ic_launcher);
    } else {
        setBackgroundColor(0xff00ffff);

        mImageView.setImageResource(R.mipmap.ic_launcher_round);
    }

    mTextView.setText(String.format(Locale.CHINA,
"%s%d: %s", getClass().getSimpleName(),
cell.pos, cell.optParam("text")));
}

@CellRender
public void postUnBindView(BaseCell cell) {
}

}
```

以上两种方式开发的组件在页面初始化的时候调用 TangramBuilder.InnerBuilder的**registerCell(String type, Class viewClz)**方法进行注册，也就是2.4节的第一种注册方式，代码如下：

```
builder.registerCell("InterfaceCell",
CustomInterfaceView.class);
builder.registerCell("AnnotationCell",
CustomAnnotationView.class);
```

意思是类型为“**InterfaceCell**”的组件渲染时会被绑定到 **CustomInterfaceView**的实例上，类型为“**AnnotationCell**”的组件渲染时会被绑定到**CustomAnnotationView**的实例上，这种方式注册的组件使用通用的组件模型BaseCell。

在自定义 View 里访问 json 数据：

组件的View对应于一个统一的model，类型是BaseCell，要在View里访问json数据，BaseCell提供了以下方法：

```
public boolean hasParam(String key)
public Object optParam(String key)
public long optLongParam(String key)
public int optIntParam(String key)
public String optStringParam(String key)
public double optDoubleParam(String key)
public boolean optBoolParam(String key)
public JSONObject optJsonObjectParam(String key)
public JSONArray optJsonArrayParam(String key)
```

这些方法都会先访问BaseCell里持有的原始json数据，同时支持访问style节点下的属性。

例如上面示例代码的第46~47行中使用的代码`cell.optParam("text")`，它的含义就是取json数据中key为“text”的字段。

3.2.2 自定义 model 开发组件

采用通用的 model 开发组件，只需要写 View 就可以了，然而需要在每次绑定数据的时候都要取原始 json 数据解析一下字段。有时候我们会有一些通用的业务字段定义，每个组件里重复解析会让代码显得冗余，因此也提供了注册自定义 model 的兼容模式开发组件。这个时候就需要写自定义 model 和自定义 View 两部分了。

(1) 自定义 model 开发

- 实现一个自定义 model 类，继承自 BaseCell。
- 实现以下几个方法：

```
/** 解析数据业务数据，可以将解析值缓存到成员变量里 */
public void parsewith(JSONObject data)
/** 解析数据样式数据，可以将解析值缓存到成员变量里 */
public void parsestyle(@Nullable JSONObject data)
/** 绑定数据到自定义 View */
public void bindview(@NonNull V view)
/** 绑定数据到 View 之后，可选实现 */
public void postBindview(@NonNull V view)
/** 校验原始数据，检查组件的合法性 */
public boolean isValid()
```

示例代码：

```
public class CustomCell extends BaseCell<CustomCellView>
{
    private String imageUrl;
    private String text;

    @Override
    public void parsewith(@NonNull JSONObject data,
@NonNull MVHelper resolver) {
        try {
            if (data.has("imageUrl")) {
                imageUrl = data.getString("imageUrl");
            }
            if (data.has("text")) {
                text = data.getString("text");
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void bindview(@NonNull CustomCellView view) {
        if (pos % 2 == 0) {
            view.setBackgroundColor(0xffff00ff);
        } else {
```

```
        view.setBackgroundColor(0xffffffff00);
    }
    view.setImageUrl(imageurl);
    view.setText(view.getClass().getSimpleName() +
pos + ":" + text);
}
}
```

(2) 自定义 View 开发

类似于3.2.1节，示例代码：

```
public class CustomCellview extends LinearLayout {
    private ImageView mImageView;
    private TextView mTextView;

    public CustomCellview(Context context) {
        super(context);
        init();
    }

    public CustomCellview(Context context, @Nullable
AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    public CustomCellview(Context context, @Nullable
AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }

    private void init() {
        setOrientation(VERTICAL);
        setGravity(Gravity.CENTER);
        int padding = Utils.dip2px(getContext(), 10);
        setPadding(padding, padding, padding, padding);
    }
}
```

```
mImageView = new ImageView(getContext());
addView(mImageView, utils.dip2px(getContext(), 110), utils.dip2px(getContext(), 72));
mTextView = new TextView(getContext());
mTextView.setPadding(0, padding, 0, 0);
addView(mTextView, LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
}

public void setImageUrl(String url) {
Glide.with(this).load(url).into(mImageView);
}

public void setText(String text) {
mTextView.setText(text);
}
}
```

这种方式开发的组件在页面初始化的时候调用

TangramBuilder.InnerBuilder的**registerCell(int type, @NonNull Class<? extends BaseCell> cellClz, @NonNull Class viewClz)**方法进行注册，也就是2.4节的第二种注册方式，代码如下：

```
builder.registerCell("CustomCell", CustomCell.class,
CustomCellview.class);
```

意思是类型为“**CustomCell**”的组件使用自定义的组件模型**CustomCell**，在渲染时会被绑定到**CustomCellView**的实例上。

也可采用2.4节的第三种注册方式，即注册绑定组件类型、自定义model、自定义ViewHolder，示例代码：

```
builder.registerCell("HolderCell",
CustomHolderCell.class,
new ViewHolderCreator<>(R.layout.item_holder,
CustomViewHolder.class, TextView.class));
```

意思是类型为“**HolderCell**”的组件使用自定义的组件模型**CustomHolderCell**，在渲染时以**R.layout.item_holder**为布局创建类型为**TextView**的 view，并绑定到类型为**CustomViewHolder**的 viewHolder 上，组件数据被绑定到**TextView**的实例上。

CustomHolderCell.java的代码如下：

```
public class CustomHolderCell extends BaseCell<TextView> {
    @Override
    public void bindView(@NonNull TextView view) {
        if (pos % 2 == 0) {
            view.setBackgroundColor(0xff000fff);
        } else {
            view.setBackgroundColor(0xfffffff000);
        }
        view.setText(String.format(Locale.CHINA, "%s%d: %s",
                getClass().getSimpleName(), pos,
                optParam("text")));
    }
}
```

item_holder.xml的代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/title"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="130dp"
    android:gravity="center"
    android:textColor="#999999"
    android:textSize="12sp"
    android:textStyle="bold"/>
```

CustomViewHolder.java的代码如下：

```
public class CustomViewHolder extends  
viewHolderCreator.ViewHolder {  
    public TextView textView;  
  
    public CustomViewHolder(Context context) {  
        super(context);  
    }  
  
    @Override  
    protected void onRootViewCreated(View view) {  
        textView = (TextView) view;  
    }  
}
```

一般情况下，使用上一节和这一节的第一种注册方式这两种方式注册组件即可。

3.2.3 json数据开发

下面针对上面开发的几种组件，根据3.1节的格式写一个简单的json示例。

在项目的assets文件夹下（没有的话自己建一个）新建**data.json**，代码如下：

```
[  
{  
    "type": "container-oneColumn",  
    "items": [  
        {  
            "type": "InterfaceCell",  
            "text": "接口方式自定义View"  
        },  
        {  
            "type": "AnnotationCell",  
            "text": "注解方式自定义View"  
        },  
        {  
            "type": "CustomCell",  
            "text": "自定义View"  
        }  
    ]  
}
```

```
        "text": "自定义model",
        "imageUrl" :
"https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-
110-72.png"
    },
    {
        "type": "HolderCell",
        "text": "自定义model和viewHolder"
    }
]
```

运行APP，显示效果如下：



Tangram Demo



CustomInterfaceView0: 接口方式自定义View



CustomAnnotationView1: 注解方式自定义View



CustomCellView2: 自定义model

CustomHolderCell3: 自定义model和ViewHolder

3.1.3 处理业务逻辑

3.3.1 处理点击

此时所有组件可以显示出来了，但是还没有点击事件。组件 View 的点击处理，可以实现**SimpleClickSupport**，然后在组件自定义的 View 内部调用**setOnClickListener(cell);**，那么组件的点击行为会被回调到**SimpleClickSupport**里统一处理。

当然使用**SimpleClickSupport**只是一种推荐的使用方式，我们选择自定义的点击处理也是可以的。

选用**SimpleClickSupport**的时候有几个注意点：

- 建议开启优化模式 —— **setOptimizedMode(true)**，这样点击会被统一回调到**public void defaultClick(View v, BaseCell cell, int pos)**方法里，我们就可以根据组件类型、View的类型即组件类型做点击处理。
- 如果不开启优化模式，**SimpleClickSupport**内部会跟进被点击的组件类型和 View 的类型来做路由，它要求我们在**SimpleClickSupport**的实现类里为每个组件的点击提供以 **onClickXXX** 或者 **onXXXClick** 为命名规范的点击处理方法，并且参数列表是**View targetView, BaseCell cell, int type**或者**View targetView, BaseCell cell, int type, Map<String, Object> params**。这种方式效率会低一些，针对采用自定义 model 开发组件的时候有用。

那么我们新建一个CustomClickSupport类，代码如下：

```
public class CustomClickSupport extends SimpleClickSupport {
    public CustomClickSupport() {
        setOptimizedMode(true);
    }

    @Override
    public void defaultClick(View targetView, BaseCell cell, int eventType) {
        Toast.makeText(targetView.getContext(),
                      "您点击了组件， type=" + cell.stringType + ", pos=" + cell.pos, Toast.LENGTH_SHORT).show();
    }
}
```

在**CustomInterfaceView.java**和**CustomAnnotationView.java**的**cellInitiated()**方法中添加如下代码：

```
public void cellInitiated(BaseCell cell) {
    setOnClickListener(cell);
}
```

在**CustomCell.java**的**bindView()**方法中添加如下代码：

```
@Override
public void bindView(@NonNull CustomCellView view) {
    ...
    view.setOnClickListener(this);
}
```

在**CustomHolderCell.java**的**bindView()**方法中添加如下代码：

```
@Override
public void bindView(@NonNull TextView view) {
    ...
    view.setOnClickListener(this);
}
```

最后如2.6节在初始化的时候注册一下，代码如下：

```
engine.addSimpleClickSupport(new CustomClickSupport());
```

运行一下APP，点击每个组件，就会弹出toast提示了。

3.3.2 处理曝光

所谓曝光，就是被 RecyclerView 的 Adapter 绑定数据的那个时候，也就是某个Item即将滑动到屏幕范围内的时候。在这个时候业务上如果需要有一些处理，就需要实现**ExposureSupport**类。它定义了3个层面的曝光接口，一是曝光布局，二是曝光组件整体区域，三是曝光组件局部区域。我们实现它的子类，并针对这三个层面的曝光做分别的实现。分别说明如下：

(1) 布局的整体曝光

需要重写回调接口方法**public abstract void onExposure(@NonNull Card card, int offset, int position);**。新建**CustomExposureSupport.java**，代码如下：

```
public class CustomExposureSupport extends ExposureSupport {
    private static final String TAG =
"CustomExposureSupport";

    public CustomExposureSupport() {
        setOptimizedMode(true);
    }

    @Override
    public void onExposure(@NonNull Card card, int offset, int position) {
        Log.d(TAG, "onExposure: card=" +
card.getClass().getSimpleName() + ", offset=" + offset +
", position=" + position);
    }
}
```

然后如2.6节在初始化的时候注册一下，代码如下：

```
engine.addExposureSupport(new CustomExposureSupport());
```

为了方便说明，我们将**data.json**改成如下代码：

```
[  
  {  
    "type": "container-oneColumn",  
    "items": [  
      {  
        "type": "InterfaceCell",  
        "text": "接口方式自定义view"  
      },  
      {  
        "type": "InterfaceCell",  
        "text": "接口方式自定义view"  
      }  
    ]  
  },  
  {  
    "type": "container-oneColumn",  
    "items": [  
      {  
        "type": "AnnotationCell",  
        "text": "注解方式自定义view"  
      },  
      {  
        "type": "AnnotationCell",  
        "text": "注解方式自定义view"  
      }  
    ]  
  },  
  {  
    "type": "container-oneColumn",  
    "items": [  
      {  
        "type": "AnnotationCell",  
        "text": "注解方式自定义view"  
      }  
    ]  
  }]
```

```
        "type": "CustomCell",
        "text": "自定义model",
        "imageUrl":
"https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-
110-72.png"
    },
    {
        "type": "CustomCell",
        "text": "自定义model",
        "imageUrl":
"https://gw.alicdn.com/tafs/TB1vqF.PpXXXXaRaXXXXXXXXXX-
110-72.png"
    }
]
},
{
    "type": "container-oneColumn",
    "items": [
        {
            "type": "HolderCell",
            "text": "自定义model和ViewHolder"
        },
        {
            "type": "HolderCell",
            "text": "自定义model和ViewHolder"
        }
    ]
}
]
```

运行APP，当我们滚到每一个布局的时候，就会打印一条日志，全部日志如下：

```
D/CustomExposureSupport: onExposure:  
card=SingleColumnCard, offset=0, position=0  
D/CustomExposureSupport: onExposure:  
card=SingleColumnCard, offset=0, position=2  
D/CustomExposureSupport: onExposure:  
card=SingleColumnCard, offset=0, position=4  
D/CustomExposureSupport: onExposure:  
card=SingleColumnCard, offset=0, position=6
```

(2) 组件的整体曝光

- 建议开启优化模式——**setOptimizedMode(true)**，这样曝光接口被统一回调到**public void defaultExposureCell(@NonNull View targetView, @NonNull BaseCell cell, int type)**方法里，我们可以根据组件类型、View的类型即组件类型做曝光处理。
- 如果不开启优化模式，**ExposureSupport**内部会跟进被点击的组件类型和 View 的类型来做路由，它要求我们在**ExposureSupport**的实现类里为每个组件的点击提供以 **onExposureXXX** 或者 **onXXXExposure** 为命名规范的点击处理方法，并且参数列表是**View targetView, BaseCell cell, int type**。这种方式效率会低一些，针对采用自定义 model 开发组件的时候有用。

在**CustomExposureSupport.java**里添加如下代码：

```
@Override  
public void defaultExposureCell(@NonNull View targetView,  
@NonNull BaseCell cell, int type) {  
    Log.d(TAG, "defaultExposureCell: targetview=" +  
targetView.getClass().getSimpleName() + ", pos=" +  
cell.pos + ", type=" + type);  
}
```

运行APP，当我们滚到每一个组件的时候，就会打印一条日志，全部日志如下：

```
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomInterfaceView, pos=0, type=0  
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomInterfaceView, pos=1, type=1  
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomAnnotationView, pos=0, type=0  
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomAnnotationView, pos=1, type=1  
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomCellView, pos=0, type=0  
D/CustomExposureSupport: defaultExposureCell:  
targetView=CustomCellView, pos=1, type=1  
D/CustomExposureSupport: defaultExposureCell:  
targetView=AppCompatTextView, pos=0, type=0  
D/CustomExposureSupport: defaultExposureCell:  
targetView=AppCompatTextView, pos=1, type=1
```

(3) 组件的局部区域曝光

- 建议开启优化模式——`setOptimizedMode(true)`, 这样曝光接口被统一回调到`public void defaultTrace(@NonNull View targetView, @NonNull BaseCell cell, int type)`方法里, 我们可以根据组件类型、View的类型即组件类型做曝光处理。
- 如果不开启优化模式, `ExposureSupport`内部会跟进被点击的组件类型和 View 的类型来做路由, 它要求我们在`ExposureSupport`的实现类里为每个组件的点击提供以 `onTraceXXX` 或者 `onXXXTrace` 为命名规范的点击处理方法, 并且参数列表是`View targetView, BaseCell cell, int type`。这种方式效率会低一些, 针对采用自定义 model 开发组件的时候有用。

前两个层面的曝光调用都是框架层调用, 而组件局部曝光, 则需要我们在组件逻辑里自行调用。使用方式如下:

```
ExposureSupport exposureSupport =  
serviceManager.getService(ExposureSupport.class);  
if (exposureSupport != null) {  
    exposureSupport.onTrace(view, cell, type);  
}
```

那么，我们在**CustomInterfaceView.java**和
CustomAnnotationView.java的**cellInitiated()**方法中添加如下代码：

```
public void cellInitiated(BaseCell cell) {  
    ...  
    if (cell.serviceManager != null) {  
        ExposureSupport exposureSupport =  
cell.serviceManager.getService(ExposureSupport.class);  
        if (exposureSupport != null) {  
            exposureSupport.onTrace(this, cell,  
cell.type);  
        }  
    }  
}
```

然后在**CustomCell.java**的**bindView()**方法中添加如下代码：

```
@Override  
public void bindView(@NonNull CustomCellView view) {  
    ...  
    if (serviceManager != null) {  
        ExposureSupport exposureSupport =  
serviceManager.getService(ExposureSupport.class);  
        if (exposureSupport != null) {  
            exposureSupport.onTrace(view, this, type);  
        }  
    }  
}
```

在**CustomHolderCell.java**的**bindView()**方法中添加如下代码：

```
@Override  
public void bindview(@NonNull TextView view) {  
    ...  
    if (serviceManager != null) {  
        ExposureSupport exposureSupport =  
            serviceManager.getService(ExposureSupport.class);  
        if (exposureSupport != null) {  
            exposureSupport.onTrace(view, this, type);  
        }  
    }  
}
```

最后在**CustomExposureSupport.java**里添加如下代码：

```
@Override  
public void defaultTrace(@NonNull View targetView,  
@NonNull BaseCell cell, int type) {  
    Log.d(TAG, "defaultTrace: targetView=" +  
        targetView.getClass().getSimpleName() + ", pos=" +  
        cell.pos + ", type=" + type);  
}
```

运行APP，当我们滚到每一个组件的时候，就会打印一条日志。与（2）不同的是，（2）是在第一次曝光的时候调用，而（3）是在每次曝光的时候都会调用。全部日志如下：

```
D/CustomExposureSupport: defaultTrace:  
targetView=CustomInterfaceView, pos=0, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=CustomInterfaceView, pos=1, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=CustomAnnotationView, pos=0, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=CustomAnnotationView, pos=1, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=CustomCellView, pos=0, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=CustomCellView, pos=1, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=AppCompatTextView, pos=0, type=0  
D/CustomExposureSupport: defaultTrace:  
targetView=AppCompatTextView, pos=1, type=0
```

3.3.3 异步加载数据

有时 Tangram 的页面的数据无法一次性返回，有些区块布局内的数据需要异步加载、甚至分页加载。Tangram 里内置了封装了异步加载的逻辑，需要各个层面配合完成，这里加以说明：

(1) 布局model 的load, loadParams, loadType, hasMore

- load 是接口名称，表示这个布局需要执行异步加载的接口。
- loadParams 是异步加载接口的常规参数字典，需要在调用接口时透传。
- loadType 是异步加载的方式，-1表示需要异步加载，1表示需要异步加载且有分页。
- hasMore 与 loadType 配合，当 loadType = 1 的时候表示分页是否结束。

示例代码：

```
{  
    "id": "Shop",  
    "load": "queryShop",  
    "loadType": "-1",  
    "type": "container-oneColumn"  
}
```

(2) **setPreLoadNumber(int preLoadNumber)**

调用**TangramEngine**上的**setPreLoadNumber(int preLoadNumber)**方法，设置触发卡片预加载的时机，默认 `preLoadNumber` 是5，表示在滑动过程中，提前去触发可见范围之外5块布局以内的异步加载逻辑。可以通过这个接口调整预加载的范围。

(3) **onScrolled()**

在recyclerView 的 `onScrollListener`里调用**TangramEngine**上的 `onScrolled()`方法，触发预加载的逻辑，代码如2.8节。

(4) **CardLoadSupport与AsyncLoader, AsyncPageLoader**

在配置完上述异步加载的基础设置之后，提供一个自定义的 **CardLoadSupport**服务，该服务需要提供一个自定义的**AsyncLoader**和**AsyncPageLoader**。

AsyncLoader的**loadData(final Card card, @NonNull final LoadedCallback loadedCallback)**方法回调是卡片异步加载的入口。加载完成之后通过**LoadedCallback**的回写接口告知布局加载是否完成。

AsyncPageLoader的**loadData(int page, @NonNull Card card, @NonNull final LoadedCallback callback)**方法是回调卡片分页加载的入口。加载完成之后通过**LoadedCallback**的回写接口告知布局加载是否完成，是否还有下一页。

代码如下：

```
CardLoadSupport mCardLoadSupport = new  
CardLoadSupport(new AsyncLoader() {  
    @Override
```

```
public void loadData(final Card card, @NonNull final LoadedCallback callback) {
    //...
}
}, new AsyncPageLoader() {

    @Override
    public void loadData(int page, @NonNull Card card,
                         @NonNull LoadedCallback
callback) {
    //...
}
});

CardLoadSupport.setInitialPage(1);
engine.addCardLoadSupport(mCardLoadSupport);
```

AsyncLoader加载成功之后回调示例代码：

```
List<BaseCell> cells =
engine.parseComponent(jsonObject.optJSONArray("items"));
callback.finish(cells);
```

AsyncPageLoader加载成功之后回调示例代码：

```
List<BaseCell> cells =
engine.parseComponent(jsonObject.optJSONArray("items"));
callback.finish(cells, itemHashMore);
```

我们就不在demo实现了，有兴趣的读者可以去深入研究。

更多组件高级用法，例如使用定时器等，可参看

<http://tangram.pingguohe.net/docs/android/use-timer>

3.1.4 布局（卡片）

Tangram的强大之处就在于可以在json中混合各种类型的布局，其主要职责是负责对组件进行布局。Tangram 内置了一系列布局能力对组件进行布局，包括流式布局、瀑布流布局、吸顶布局、悬浮布局、轮播布局等等，基本上常见的布局方式都可以覆盖到。

下面对各种布局分别进行阐述。

3.4.1 流式（网格）布局

流式布局是最常用的布局。[详细说明](#)

| type | 对应类型 |
|-----------------------|---------|
| container-oneColumn | 单列(一排一) |
| container-twoColumn | 双列 |
| container-threeColumn | 三列 |
| container-fourColumn | 四列 |
| container-fiveColumn | 五列 |
| container-flow | N列 |

3.4.2 一拖N布局

左边一个大的，右边N个小的，可调整比例。[详细说明](#)

有三种样式：

- 左边一个，右边上面一个下面一个
- 左边一个，右边上面一个下面两个
- 左边一个，右边上面一个下面三个

会根据数据的数量自动区分

| type | 对应类型 |
|--------------------|---------|
| container-onePlusN | 一拖2/3/4 |

3.4.3 浮动布局

可拖动，自动吸边。[详细说明](#)

| type | 对应类型 |
|-----------------|------|
| container-float | 浮标 |

3.4.4 固定布局

固定在某个位置，不可拖动。[详细说明](#)

| type | 对应类型 |
|---------------------|------------------------|
| container-fix | 固定顶部或者底部，根据属性指定 |
| container-scrollFix | 滚动固定(滚动到某个布局的时候，出现并固定) |

3.4.5 吸顶布局

碰到Tangram的顶端或底端就吸住。[详细说明](#)

| type | 对应类型 |
|------------------|--------------|
| container-sticky | 吸顶或吸底，根据属性指定 |

3.4.6 轮播滚动布局

适用于Banner的场景，按页可自动滚动，循环滚动。[详细说明](#)

| type | 对应类型 |
|------------------|------|
| container-banner | 轮播 |

3.4.7 横向滚动布局

适用于做线性的滚动，而不是Banner一页一页的滚动。[详细说明](#)

| type | 对应类型 |
|------------------|----------------------|
| container-scroll | 线性滚动，不像轮播一样具有一页一页的效果 |

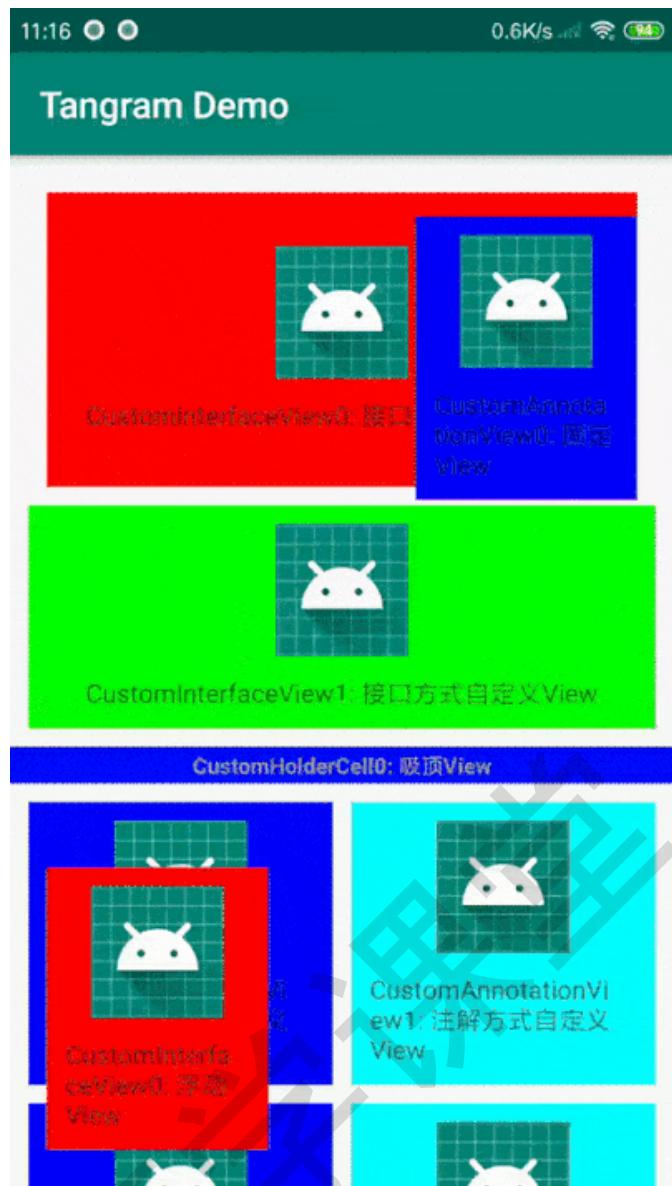
3.4.8 瀑布流布局

[详细说明](#)

| type | 对应类型 |
|---------------------|------|
| container-waterfall | 瀑布流 |

3.1.5 示例

基于上面已经实现的组件，我实现了一个示例，包含了上面列出的所有类型的布局，效果如下：



修改的**data.json**文件请见：

<https://github.com/jimmysuncpt/TangramDemo/blob/master/app/src/main/assets/data.json>，大家可以把最后的container-oneColumn去掉，这个是后面用来演示VirtualView的。

为了演示设置背景，我还新增了一个无背景的类型，无背景的自定义View代码请见：

<https://github.com/jimmysuncpt/TangramDemo/blob/master/app/src/main/java/com/jimmysun/tangramdemo/tangram/NoBackgroundView.java>

别忘了在初始化的时候注册一下：

```
builder.registerCell("NoBackground",  
    NoBackgroundview.class);
```

完整示例请见：<https://github.com/jimmysuncpt/TangramDemo>

以上，我们学习了Tangram的概念、使用步骤以及组件与布局的开发。

在 Tangram 体系里，页面结构可以通过配置动态更新，然而业务组件是通过 Java 代码实现的，无法动态更新。为了解决业务组件的动态更新，阿里后来又提出了VirtualView，具体教程可参见[Android动态界面开发框架VirtualView使用完整教程](#)。

第四章 网络通信篇

第一节 网络协议

- WebSocket 原理

1.1 看完让你彻底理解 WebSocket 原理，附完整的实战代码（包含前端和后端）

1.1.1前言

最近有同学问我有没有做过在线咨询功能。同时，公司也刚好让我接手一个 IM 项目。所以今天抽时间记录一下最近学习的内容。本文主要剖析了 WebSocket 的原理，以及附上一个完整的聊天室实战 Demo（包含前端和后端，代码下载链接在文末）。

1.1.2WebSocket 与 HTTP

WebSocket 协议在2008年诞生，2011年成为国际标准。现在所有浏览器都已经支持了。WebSocket 的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话。

HTTP 有 1.1 和 1.0 之说，也就是所谓的 keep-alive，把多个 HTTP 请求合并为一个，但是 Websocket 其实是一个新协议，跟 HTTP 协议基本没有关系，只是为了兼容现有浏览器，所以在握手阶段使用了 HTTP。

下面一张图说明了 HTTP 与 WebSocket 的主要区别：

WebSocket 的其他特点：

- 建立在 TCP 协议之上，服务器端的实现比较容易。
- 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
- 数据格式比较轻量，性能开销小，通信高效。
- 可以发送文本，也可以发送二进制数据。
- 没有同源限制，客户端可以与任意服务器通信。
- 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

1.1.3 WebSocket 是什么样的协议，具体有什么优点

首先，WebSocket 是一个持久化的协议，相对于 HTTP 这种非持久的协议来说。简单的举个例子吧，用目前应用比较广泛的 PHP 生命周期来解释。

HTTP 的生命周期通过 Request 来界定，也就是一个 Request 一个 Response，那么在 HTTP1.0 中，这次 HTTP 请求就结束了。

在 HTTP1.1 中进行了改进，使得有一个 keep-alive，也就是说，在一个 HTTP 连接中，可以发送多个 Request，接收多个 Response。但是请记住 Request = Response，在 HTTP 中永远是这样，也就是说一个 Request 只能有一个 Response。而且这个 Response 也是被动的，不能主动发起。

你 BB 了这么多，跟 WebSocket 有什么关系呢？好吧，我正准备说 WebSocket 呢。

首先 WebSocket 是基于 HTTP 协议的，或者说借用了 HTTP 协议来完成一部分握手。

首先我们来看个典型的 WebSocket 握手

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

熟悉 HTTP 的童鞋可能发现了，这段类似 HTTP 协议的握手请求中，多了这么几个东西。

```
Upgrade: websocket
Connection: Upgrade
```

这个就是 WebSocket 的核心了，告诉 Apache、Nginx 等服务器：注意啦，我发起的请求要用 WebSocket 协议，快点帮我找到对应的助理处理~而不是那个老土的 HTTP。

```
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

首先，`Sec-WebSocket-Key` 是一个 Base64 encode 的值，这个是浏览器随机生成的，告诉服务器：泥煤，不要忽悠我，我要验证你是不是真的是 WebSocket 助理。

然后，`Sec-WebSocket-Protocol` 是一个用户定义的字符串，用来区分同 URL 下，不同的服务所需要的协议。简单理解：今晚我要服务 A，别搞错啦~

最后，`Sec-WebSocket-Version` 是告诉服务器所使用的 WebSocket Draft (协议版本)，在最初的时候，WebSocket 协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么 Firefox 和 Chrome 用的不是一个版本之类的，当初 WebSocket 协议太多可是一个大难题。。不过现在还好，已经定下来啦~大家都使用同一个版本：服务员，我要的是13岁的噢→_→

然后服务器会返回下列东西，表示已经接受到请求，成功建立 WebSocket 啦！

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUKAGmm5OPpG2HaGwk=
Sec-WebSocket-Protocol: chat
```

这里开始就是 HTTP 最后负责的区域了，告诉客户，我已经成功切换协议啦～

```
Upgrade: websocket
Connection: Upgrade
```

依然是固定的，告诉客户端即将升级的是 WebSocket 协议，而不是 mozillasocket, lurnarsocket 或者 shitsocket。

然后， Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key 。 服务器：好啦好啦，知道啦，给你看我的 ID CARD 来证明行了吧。

后面的， Sec-WebSocket-Protocol 则是表示最终使用的协议。

至此， HTTP 已经完成它所有工作了，接下来就是完全按照 WebSocket 协议进行了。

1.1.4WebSocket 的作用

在讲 WebSocket 之前，我就顺带着讲下 ajax 轮询 和 long poll 的原理。

4-1、 ajax 轮询

ajax 轮询的原理非常简单，让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息。

场景再现：

```
客户端：啦啦啦，有没有新信息(Request)
```

服务端：没有（Response）

客户端：啦啦啦，有没有新信息（Request）

服务端：没有。。。 （Response）

客户端：啦啦啦，有没有新信息（Request）

服务端：你好烦啊，没有啊。。。 （Response）

客户端：啦啦啦，有没有新消息（Request）

服务端：好啦好啦，有啦给你。 （Response）

客户端：啦啦啦，有没有新消息（Request）

服务端：。。。。。没。。。没。。。没有（Response） -- loop

4-2、long poll

long poll 其实原理跟 ajax 轮询差不多，都是采用轮询的方式，不过采取的是阻塞模型（一直打电话，没收到就不挂电话），也就是说，客户端发起请求后，如果没消息，就一直不返回 Response 给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

场景再现：

客户端：啦啦啦，有没有新信息，没有的话就等有了才返回给我吧
(Request)

服务端：额。。。等待到有消息的时候。。。来 给你（Response）

客户端：啦啦啦，有没有新信息，没有的话就等有了才返回给我吧
(Request) -loop

从上面可以看出其实这两种方式，都是在不断地建立HTTP连接，然后等待服务端处理，可以体现HTTP协议的另外一个特点，被动性。

何为被动性呢， 其实就是， 服务端不能主动联系客户端， 只能有客户端发起。

从上面很容易看出来， 不管怎么样， 上面这两种都是非常消耗资源的。

ajax轮询 需要服务器有很快的处理速度和资源。 long poll 需要有很高的并发， 也就是说同时接待客户的能力。

所以 ajax轮询 和 long poll 都有可能发生这种情况。

客户端： 啦啦啦啦， 有新信息么？

服务端： 正忙，请稍后再试（503 Server Unavailable）

客户端： 好吧， 啦啦啦， 有新信息么？

服务端： 正忙，请稍后再试（503 Server Unavailable）

4-3、 WebSocket

通过上面这两个例子， 我们可以看出， 这两种方式都不是最好的方式， 需要很多资源。

一种需要更快的速度， 一种需要更多的‘电话’。这两种都会导致‘电话’的需求越来越高。

哦对了， 忘记说了 HTTP 还是一个无状态协议。通俗的说就是， 服务器因为每天要接待太多客户了， 是个健忘鬼， 你一挂电话， 他就把你东西全忘光了， 把你的东西全丢掉了。你第二次还得再告诉服务器一遍。

所以在这种情况下出现了 WebSocket 。他解决了 HTTP 的这几个难题。

首先， 被动性， 当服务器完成协议升级后（HTTP->Websocket）， 服务端就可以主动推送信息给客户端啦。所以上面的情景可以做如下修改。

客户端： 啦啦啦， 我要建立websocket协议， 需要的服务： chat，
websocket协议版本： 17（HTTP Request）

服务端： ok， 确认， 已升级为websocket协议（HTTP Protocols
Switched）

客户端：麻烦你有信息的时候推送给我噢。。

服务端：ok，有的时候会告诉你的。

服务端：ba1ab1ab1ab1ab1a

服务端：ba1ab1ab1ab1ab1a

服务端：哈哈哈哈哈哈啊哈哈哈哈

服务端：笑死我了哈哈哈哈哈哈哈

这样，只需要经过一次 HTTP 请求，就可以做到源源不断的信息传送了。

1.1.5 实战代码

[本文的更新源托管于 GitHub](#)

参考文档：

[php socket 文档](#)

[js 的 WebSocket 文档](#)

前端代码：<https://github.com/nnngu/WebSocketDemo-js>

后端代码：<https://github.com/nnngu/WebSocketDemo-php>

运行步骤：

1. 在终端打开 `WebSocketDemo-php` 目录，执行 `php -q server.php`
2. 用浏览器访问 `WebSocketDemo-js` 目录里面的 `index.html`

第五章 架构设计篇

第一节 MVP 架构设计

1.1 高级MVP架构封装演变全过程

1.1.1 概述

本文不会讲解什么是MVP，如果还不太了解MVP请自行查阅资料，本文重点是封装一个高级MVP架构，会详细的讲解如何一步步从无到有的封装成一个高级MVP架构过程。

众所周知普通的MVP模式可能存在内存泄露、代码冗余、界面意外关闭后在重建数据缓存等问题，本文最终封装的成果为一一解决这些问题，而且在使用过程中尽量做到使用简单而且可扩展，当然本文也只是提供了一种封装思路而已，如果不能满足你的需求还可以自行再进行扩展。

如果你觉得你不想看整个实现思路可以直接看最后的源码，望给点个star
鼓励一下

GitHub地址：

<https://github.com/ljqloveyou123>

文章会以5个部分来整体优化封装MVP，也是一个从无到有的过程

- 一、不使用MVP的代码
- 二、最简单的MVP实现
- 三、解决MVP内存泄露
- 四、简单抽象-解决MVP代码冗余
- 五、高级抽象-使用注解、工厂模式、代理模式、策略模式整体解决代码冗余、内存泄露、Presenter生命周期以及数据存储问题

不废话了，进入正题。

场景：假如界面有一个按钮，点击后请求数据，然后成功后将返回的数据设置到一个Textview中

1.1.2 不使用MVP的代码

一、不使用MVP的代码，一般我们会这么写

```
public class MainActivity extends AppCompatActivity {
```

```
@Fieldview(R.id.tv_text)
private TextView textView;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    viewFind.bind(this);
}
//按钮点击事件
public void request(View view) {
    clickRequest("101010100");
}
//发起请求
public void clickRequest(final String cityId) {
    //请求接口
    Retrofit retrofit = new Retrofit.Builder()
        //代表root地址
        .baseUrl("http://www.weather.com.cn/")
        .addConverterFactory(ScalarsConverterFactory.create())
        .addConverterFactory(GsonConverterFactory.create())
        .build();
    ApiService apiService =
    retrofit.create(ApiService.class);
    //请求
    Call<WeatherBean> weatherBeanCall =
    apiService.requestWeather(cityId);
    weatherBeanCall.enqueue(new Callback<WeatherBean>()
    {
        @Override
        public void onResponse(Call<WeatherBean> call, Response<WeatherBean> response) {
            //请求成功
            textView.setText(response.body().getWeatherInfo().toString());
        }
    });
}
```

```
    @Override
    public void onFailure(Call<weatherBean> call,
    Throwable t) {
        //请求失败
    }
});
```

上面的代码是最原始的写法，下面我们使用最简单的MVP模式来改造这个代码。

1.1.3最简单的MVP实现

思路如下：

- 1、首先我们先定义一个接口，用来规定针对这个界面逻辑View需要作出的动作的接口。
- 2、让Activity实现这个接口中的方法，也就是V层
- 3、创建一个类，用来封装之前的网络请求过程，也就是M层
- 4、再创建一个类，用来处理M层和V层之间的通信，也就是P层

下面来实现一下：

- 1、首先我们先定义一个接口，用来规定针对这个界面逻辑View需要作出的动作的接口。

```
/**
 * @author 刘稼旗
 * @date 2017/11/16
 * @description V层接口，定义V层需要作出的动作的接口
 */
public interface RequestView1 {
    //请求时展示加载
    void requestLoading();
    //请求成功
    void resultSuccess(WeatherBean result);
    //请求失败
    void resultFailure(String result);
}
```

2、让Activity实现这个接口中的方法，也就是V层

```
/**
 * 第二步：对应demo1
 * 2. 最简单的MVP模式，
 * 1. Activity需要实现V层接口
 * 2. Presenter需要持有V层引用和M层引用
 * 3. 在实现类view中创建presenter
 */
public class MainActivity extends AppCompatActivity
implements RequestView1 {
    @BindView(R.id.tv_text)
    private TextView textView;
    private RequestPresenter1 presenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ViewFind.bind(this);
        //创建Presenter
        presenter = new RequestPresenter1(this);
    }
    //点击事件
    public void request(View view) {
```

```
presenter.clickRequest("101010100");
}

//请求时加载
@Override
public void requestLoading() {
    textView.setText("请求中,请稍后....");
}

//请求成功
@Override
public void resultSuccess(WeatherBean result) {
    //成功

    textView.setText(result.getWeatherInfo().toString());
}

//请求失败
@Override
public void resultFailure(String result) {
    //失败
    textView.setText(result);
}
}
```

3、创建一个类，用来封装之前的网络请求过程，也就是M层

```
/**
 * @description M层 数据层
 */
public class RequestModel {
    private static final String BASE_URL =
"http://www.weather.com.cn/";

//http://www.weather.com.cn/data/cityinfo/101010100.html
    public void request(String detailId,
Callback<WeatherBean> callback){
        //请求接口
        Retrofit retrofit = new Retrofit.Builder()
            //代表root地址
            .baseUrl(BASE_URL)
```

```
.addConverterFactory(ScalarsConverterFactory.create())
.addConverterFactory(GsonConverterFactory.create())
    .build();
    ApiService apiService =
retrofit.create(ApiService.class);
//请求
Call<WeatherBean> weatherBeanCall =
apiService.requestWeather(detailId);
weatherBeanCall.enqueue(callback);
}
}
```

4、再创建一个类，用来处理M层和V层之间的通信，也就是P层

```
/**
 * @description P层
 * 特点：需要持有M层和V层
 */
public class RequestPresenter1 {
    private final RequestView1 mRequestView;
    private final RequestModel1 mRequestMode;
    public RequestPresenter1(RequestView1 requestView) {
        this.mRequestView = requestView;
        this.mRequestMode = new RequestModel1();
    }
    public void clickRequest(final String cityId){
        //请求时显示加载
        mRequestView.requestLoading();
        //模拟耗时，可以展示出loading
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                mRequestMode.request(cityId, new
Callback<WeatherBean>() {
                    @Override
```

```
        public void  
onResponse(Call<WeatherBean> call, Response<WeatherBean>  
response) {  
  
    mRequestView.resultSuccess(response.body());  
    }  
    @Override  
    public void  
onFailure(Call<WeatherBean> call, Throwable t) {  
  
    mRequestView.resultFailure(Log.getStackTraceString(t));  
    }  
    }  
}, 1000);  
}  
}
```

好了，上面的4步就是最基本的MVP模式的使用了，可是这样写会内存泄露，因为如果在网络请求的过程中Activity就关闭了，Presenter还持有了V层的引用，也就是MainActivity，就会内存泄露。

1.1.4解决MVP内存泄露

下面就来解决这个问题，我们将P层和V层的关联抽出两个方法，一个绑定，一个解绑，在需要的时候进行绑定V层，不需要的时候进行解绑就可以了。

我们只需要修改上面Presenter中的构造代码，不需要在构造中传递V层了，然后再写一个绑定和解绑的方法，最后修改Activity创建Presenter时进行绑定，在onDestroy中进行解绑。

修改后的Presenter：

```
public class RequestPresenter2 {  
    private RequestView2 mView;  
    private RequestMode2 mMode;
```

```
public RequestPresenter2() {
    mMode = new RequestMode2();
}

public void clickRequest(final String cityId) {
    if(mview != null){
        mview.requestLoading();
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                mMode.request(cityId, new
Callback<WeatherBean>() {
                    @Override
                    public void
onResponse(Call<WeatherBean> call, Response<WeatherBean>
response) {
                        if(mview != null){
                            mview.resultSuccess(response.body());
                        }
                    }
                    @Override
                    public void
 onFailure(Call<WeatherBean> call, Throwable t) {
                        if(mview != null){
                            mview.resultFailure(Log.getStackTraceString(t));
                        }
                    }
                });
            }
        },1000);
    }
}

/**
 * 绑定
 * @param view
 */
public void attach( RequestView2 view) {
```

```
        this.mView = view;
    }
    /**
     * 解除绑定
     */
    public void detach() {
        mView = null;
    }
    /**
     * 取消网络请求
     */
    public void interruptHttp(){
        mMode.interruptHttp();
    }
}
```

修改后的MainActivity：

```
/**
 * 第三步：对应demo2
 * 上面的问题：
 * 1. 是会内存泄露，因为presenter一直持有Activity，如果一个发了一个请求，
 * 但是网络有点慢，这个时候退出Activity，那么请求回来后还是会调用
 * Activity的回调方法，这里还是因为一直持有的问题
 * 2. 如果已经退出了当前界面，这个请求也没有用了，这个时候我们可以断开请求
 * <p>
 * 解决问题：
 * 1. 增加绑定和解绑的方法来解决内存泄露和退出后还会回调的问题
 * 2、增加断开网络连接的方法
 */
public class MainActivity extends AppCompatActivity
implements RequestView2 {
    @Fieldview(R.id.tv_text)
    private TextView textView;
    private RequestPresenter2 presenter;
    @Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ViewFind.bind(this);
    presenter = new RequestPresenter2();
    presenter.attach(this);
}

public void request(View view) {
    presenter.clickRequest("101010100");
}

@Override
public void requestLoading() {
    textView.setText("请求中，请稍后....");
}

@Override
public void resultSuccess(WeatherBean result) {
    //成功

    textView.setText(result.getWeatherInfo().toString());
}

@Override
public void resultFailure(String result) {
    //失败
    textView.setText(result);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    presenter.detach();
    presenter.interruptHttp();
}

}
```

这样我们就解决了内存泄露的问题，但是这样还是不完美，应用中肯定不可能只有一个模块，每个模块都对应着一个V层和P层，那这样的话每个Presenter中都要定义绑定和解绑的方法，而Activity中对应的也要调用这绑定和解绑的两个方法，代码冗余。

1.1.5简单抽象-解决MVP代码冗余

针对这个问题我们可以抽取出一个基类的Presenter和一个基类的Activity来做这个事情，让子类不用在写这些重复的代码。但是问题又来了，既然是基类，肯定不止有一个子类来继承基类，那么也就是说子类当中定义的View接口和需要创建的Presenter都不相同，我们肯定在基类当中不能写死吧，那就使用泛型来设计。

思路：

1. 创建一个基类View，让所有View接口都必须实现，这个View可以什么都不做只是用来约束类型的
2. 创建一个基类的Presenter，在类上规定View泛型，然后定义绑定和解绑的抽象方法，让子类去实现，对外在提供一个获取View的方法，让子类直接通过方法来获取View
3. 创建一个基类的Activity，声明一个创建Presenter的抽象方法，因为要帮子类去绑定和解绑那么就需要拿到子类的Presenter才行，但是又不能随便一个类都能绑定的，因为只有基类的Presenter中才定义了绑定和解绑的方法，所以同样的在类上可以声明泛型在，方法上使用泛型来达到目的。
4. 修改Presenter和Activity中的代码，各自继承自己的基类并去除重复代码

实现步骤：

1. 创建一个基类View，让所有View接口都必须实现

```
/**  
 * @author 刘稼旗  
 * @date 2017/11/17  
 * @description 所有mvpview的父接口  
 */  
public interface IMvpBaseView4 {  
}
```

2. 创建一个基类的Presenter，在类上规定View泛型，然后定义绑定和解绑的方法，对外在提供一个获取View的方法，让子类直接通过方法来获取View使用即可

```
/**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description 指定绑定的view必须继承自IMvpBaseView4
 */
public abstract class AbstractMvpPersenter4<V extends IMvpBaseView4> {
    private V mMvpView;
    /**
     * 绑定V层
     * @param view
     */
    public void attachMvpView(V view){
        this.mMvpView = view;
    }
    /**
     * 解除绑定V层
     */
    public void detachMvpView(){
        mMvpView = null;
    }
    /**
     * 获取V层
     * @return
     */
    public V getmMvpView() {
        return mMvpView;
    }
}
```

3. 创建一个基类的Activity，声明一个创建Presenter的抽象方法，因为要帮子类去绑定和解绑那么就需要拿到子类的Presenter才行，但是又不能随便一个类都能绑定的，因为只有基类的Presenter中才定义了绑定和解绑的方法，所以同样的在类上可以声明泛型在方法上使用泛型来达到目的

```
/***
 * @description MvpActivity
 * 指定子类具体的view必须继承自IMvpBaseview4
 * 指定子类具体的Presenter必须继承自AbstractMvpPersenter4
 */
public abstract class AbstractMvpActivity<V extends
IMvpBaseview4, P extends AbstractMvpPersenter4<V>>
        extends AppCompatActivity implements
IMvpBaseview4 {
    private P presenter;
    @Override
    protected void onCreate(@Nullable Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        //创建Presenter
        if (presenter == null) {
            presenter = createPresenter();
        }
        if (presenter == null) {
            throw new NullPointerException("presenter 不能为空!");
        }
        //绑定view
        presenter.attachMvpView((V) this);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        //解除绑定
        if (presenter != null) {
            presenter.detachMvpView();
        }
    }
}
/***
 * 创建Presenter
 * @return 子类自己需要的Presenter
 */
protected abstract P createPresenter();
```

```
 /**
 * 获取Presenter
 * @return 返回子类创建的Presenter
 */
public P getPresenter() {
    return presenter;
}
}
```

4.修改Presenter和Activity中的代码，各自继承自己的基类并去除重复代码

修改后的Presenter:

```
 /**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description P层
 */
public class RequestPresenter4 extends
AbstractMvpPersenter4<RequestView4> {
    private final RequestMode4 mRequestMethod;
    public RequestPresenter4() {
        this.mRequestMethod = new RequestMode4();
    }
    public void clickRequest(final String cityId){
        //获取view
        if(getMvpView() != null){
            getMvpView().requestLoading();
        }
        //模拟网络延迟，可以显示出加载中
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                mRequestMethod.request(cityId, new
Callback<weatherBean>() {
                    @Override
```

```
        public void  
onResponse(Call<WeatherBean> call, Response<WeatherBean>  
response) {  
            //判断view是否为空  
            if(getMvpView() != null){  
  
getMvpView().resultSuccess(response.body());  
            }  
        }  
        @Override  
        public void  
onFailure(Call<WeatherBean> call, Throwable t) {  
            if(getMvpView() != null){  
  
getMvpView().resultFailure(Log.getStackTraceString(t));  
            }  
        }  
    };  
},1000);  
}  
public void interruptHttp(){  
    mRequestMethod.interruptHttp();  
}  
}  
}
```

修改后的Activity：

```
public class MainActivity extends  
AbstractMvpActivity<RequestView4, RequestPresenter4>  
implements RequestView4 {  
    @Fieldview(R.id.tv_text)  
    private TextView textView;  
    private RequestPresenter3 presenter;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```

```
    viewFind.bind(this);
}
@Override
protected RequestPresenter4 createPresenter() {
    return new RequestPresenter4();
}
//点击事件
public void request(View view) {
    getPresenter().clickRequest("101010100");
}
@Override
public void requestLoading() {
    textView.setText("请求中，请稍后....");
}
@Override
public void resultSuccess(WeatherBean result) {
    //成功
    textView.setText(result.getWeatherInfo().toString());
}
@Override
public void resultFailure(String result) {
    //失败
    textView.setText(result);
}
}
```

到这里完美了吗？没有，还可以再抽，来分析一下还有哪些不完美的地方以及如何再优化。

1.1.6 进一步的优化

高级抽象-使用注解、工厂模式、代理模式、策略模式整体解决代码冗余、内存泄露、Presenter生命周期以及数据存储问题

1. 每个子类都要重写父类创建Presenter的方法，创建一个Presenter并返回，这一步我们也可以让父类帮忙干了，怎么做呢？

我们可以采用注解的方式，在子类上声明一个注解并注明要创建的类型，剩下的事情就让父类去做了，但是父类得考虑如果子类不想这么干怎么办，那也还是不能写死吧，可以使用策略模式加工厂模式来实现，我们默认使用这种注解的工厂，但是如果子类不喜欢可以通过父类提供的一个方法来创建自己的工厂。

2.Presenter真正的创建过程，我们可以将它放到真正使用Presenter的时候再创建，这样的话可以稍微优化一下性能问题

3.界面有可能会意外销毁并重建，Activity、Fragment、View都可以在销毁的时候通过onDestroy释放一些资源并在onSaveInstanceState方法中存储一些数据然后在重建的时候恢复，但是有可能Presenter中也需要释放一些资源存储一些数据，那么上面的结构就不能满足了，我们可以给Presenter增加生命周期的方法，让Presenter和V层生命周期同步就可以做到了

4.第三步中我们又给Presenter加入了一些生命周期的方法，再加上Presenter的创建绑定和解绑的方法，那么如果我们在创建一个MvpFragment基类，或者View的基类那么这么多的代码岂不是都要copy一份吗，而且看起来也很不清晰，这里我们可以采用代理模式来优化一下。

好了下面来实现：

1.既然要采用工厂模式才创建Presenter，那么我们首先来创建一个工厂接口

```
/**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description Presenter工厂接口
 */
public interface PresenterMvpFactory<V extends BaseMvpView, P extends BaseMvpPresenter<V>> {
    /**
     * 创建Presenter的接口方法
     * @return 需要创建的Presenter
     */
    P createMvpPresenter();
}
```

2.然后我们需要创建一个默认使用注解创建的工厂，那么首先要创建一个注解

注解：

```
/**
 * @description 标注创建Presenter的注解
 */
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface CreatePresenter {
    Class<? extends BaseMvpPresenter> value();
}
```

注解工厂：

```
/**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description Presenter工厂实现类
 */
public class PresenterMvpFactoryImpl<V extends BaseMvpView, P extends BaseMvpPresenter<V>> implements
    PresenterMvpFactory<V, P> {
```

```
 /**
 * 需要创建的Presenter的类型
 */
private final Class<P> mPresenterClass;
/**
 * 根据注解创建Presenter的工厂实现类
 * @param viewClazz 需要创建Presenter的V层实现类
 * @param <V> 当前View实现的接口类型
 * @param <P> 当前要创建的Presenter类型
 * @return 工厂类
 */
public static <V extends BaseMvpView, P extends
BaseMvpPresenter<V>> PresenterMvpFactoryImpl<V, P>
createFactory(Class<?> viewClazz){
    CreatePresenter annotation =
viewClazz.getAnnotation(CreatePresenter.class);
    Class<P> aClass = null;
    if(annotation != null){
        aClass = (Class<P>) annotation.value();
    }
    return aClass == null ? null : new
PresenterMvpFactoryImpl<V, P>(aClass);
}
private PresenterMvpFactoryImpl(Class<P>
presenterClass) {
    this.mPresenterClass = presenterClass;
}
@Override
public P createMvpPresenter() {
    try {
        return mPresenterClass.newInstance();
    } catch (Exception e) {
        throw new RuntimeException("Presenter创建失败!, 检查是否声明了@CreatePresenter(xx.class)注解");
    }
}
```

3.我们说了不能写死这个工厂，那么我们需要使用者可以自定义，那么我们还需要给使用者提供一个设置的方法，我们定义一个接口提供设置工厂、获取工厂、获取Presenter的方法，然后让V层来实现这个接口，这样V层的子类就可以通过相应的方法使用了

```
/**
 * @author 刘稼旗
 * @date 2017/11/20
 * @description 代理接口
 */
public interface PresenterProxyInterface<V extends BaseMvpView, P extends BaseMvpPresenter<V>> {
    /**
     * 设置创建Presenter的工厂
     * @param presenterFactory PresenterFactory类型
     */
    void setPresenterFactory(PresenterMvpFactory<V, P> presenterFactory);

    /**
     * 获取Presenter的工厂类
     * @return 返回PresenterMvpFactory类型
     */
    PresenterMvpFactory<V, P> getPresenterFactory();

    /**
     * 获取创建的Presenter
     * @return 指定类型的Presenter
     */
    P getMvpPresenter();
}
```

4.给Presenter增加生命周期的方法

```
/**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description 所有Presenter的基类，并不强制实现这些方法，有需要在重写
 */
```

```
public class BaseMvpPresenter<V extends BaseMvpView> {  
    /**  
     * V层view  
     */  
    private V mView;  
    /**  
     * Presenter被创建后调用  
     *  
     * @param savedInstanceState 被意外销毁后重建后的Bundle  
     */  
    public void onCreatePersenter(@Nullable Bundle savedInstanceState) {  
        Log.e("perfect-mvp", "P onCreatePersenter = ");  
    }  
    /**  
     * 绑定View  
     */  
    public void onAttachMvpView(V mvpView) {  
        mView = mvpView;  
        Log.e("perfect-mvp", "P onResume");  
    }  
    /**  
     * 解除绑定View  
     */  
    public void onDetachMvpView() {  
        mView = null;  
        Log.e("perfect-mvp", "P onDetachMvpView = ");  
    }  
    /**  
     * Presenter被销毁时调用  
     */  
    public void onDestroyPersenter() {  
        Log.e("perfect-mvp", "P onDestroy = ");  
    }  
    /**  
     * 在Presenter意外销毁的时候被调用，它的调用时机和Activity、  
     * Fragment、View中的onSaveInstanceState  
     * 时机相同  
     */
```

```
* @param outState
*/
public void onSaveInstanceState(Bundle outState) {
    Log.e("perfect-mvp", "P onSaveInstanceState = ");
}

/**
 * 获取V层接口View
 *
 * @return 返回当前MvpView
 */
public V getMvpView() {
    return mView;
}
}
```

5. 创建一个代理来管理Presenter的生命周期方法

```
/**
 * @author 刘稼旗
 * @date 2017/11/20
 * @description 代理实现类，用来管理Presenter的生命周期，还有和
view之间的关联
*/
public class BaseMvpProxy<V extends BaseMvpView, P
extends BaseMvpPresenter<V>> implements
PresenterProxyInterface<V, P>{
    /**
     * 获取onSaveInstanceState中bundle的key
     */
    private static final String PRESENTER_KEY =
"presenter_key";
    /**
     * Presenter工厂类
     */
    private PresenterMvpFactory<V, P> mFactory;
    private P mPresenter;
    private Bundle mBundle;
```

```
private boolean mIsAttachedView;
public BaseMvpProxy(PresenterMvpFactory<V, P>
presenterMvpFactory) {
    this.mFactory = presenterMvpFactory;
}
/***
 * 设置Presenter的工厂类，这个方法只能在创建Presenter之前调用，也就是调用getMvpPresenter()之前，如果Presenter已经创建则不能再修改
 *
 * @param presenterFactory PresenterFactory类型
 */
@Override
public void
setPresenterFactory(PresenterMvpFactory<V, P>
presenterFactory) {
    if (mPresenter != null) {
        throw new IllegalArgumentException("这个方法只能在getMvpPresenter()之前调用，如果Presenter已经创建则不能再修改");
    }
    this.mFactory = presenterFactory;
}
/***
 * 获取Presenter的工厂类
 *
 * @return PresenterMvpFactory类型
 */
@Override
public PresenterMvpFactory<V, P>
getPresenterFactory() {
    return mFactory;
}
/***
 * 获取创建的Presenter
 *
 * @return 指定类型的Presenter
 * 如果之前创建过，而且是以外销毁则从Bundle中恢复
*/
```

```
 */
@Override
public P getMvpPresenter() {
    Log.e("perfect-mvp", "Proxy getMvpPresenter");
    if (mFactory != null) {
        if (mPresenter == null) {
            mPresenter =
mFactory.createMvpPresenter();
            mPresenter.onCreatePersenter(mBundle ==
null ? null : mBundle.getBundle(PRESENTER_KEY));
        }
    }
    Log.e("perfect-mvp", "Proxy getMvpPresenter = " +
mPresenter);
    return mPresenter;
}
/**
 * 绑定Presenter和view
 * @param mvpView
 */
public void onResume(V mvpView) {
    getMvpPresenter();
    Log.e("perfect-mvp", "Proxy onResume");
    if (mPresenter != null && !mIsAttchview) {
        mPresenter.onAttachMvpview(mvpView);
        mIsAttchview = true;
    }
}
/**
 * 销毁Presenter持有的view
 */
private void onDetachMvpview() {
    Log.e("perfect-mvp", "Proxy onDetachMvpView = ");
    if (mPresenter != null && mIsAttchview) {
        mPresenter.onDetachMvpview();
        mIsAttchview = false;
    }
}
```

```
/**
 * 销毁Presenter
 */
public void onDestroy() {
    Log.e("perfect-mvp", "Proxy onDestroy = ");
    if (mPresenter != null) {
        onDetachMvpView();
        mPresenter.onDestroyPersenter();
        mPresenter = null;
    }
}

/**
 * 意外销毁的时候调用
 * @return Bundle, 存入回调给Presenter的Bundle和当前
Presenter的id
 */
public Bundle onSaveInstanceState() {
    Log.e("perfect-mvp", "Proxy onSaveInstanceState = ");
    Bundle bundle = new Bundle();
    getMvpPresenter();
    if(mPresenter != null){
        Bundle presenterBundle = new Bundle();
        //回调Presenter
        mPresenter.onSaveInstanceState(presenterBundle);
        bundle.putBundle(PRESENTER_KEY,presenterBundle);
    }
    return bundle;
}

/**
 * 意外关闭恢复Presenter
 * @param savedInstanceState 意外关闭时存储的Bundler
 */
public void onRestoreInstanceState(Bundle
savedInstanceState) {
```

```

        Log.e("perfect-mvp", "Proxy onRestoreInstanceState
= ");
        Log.e("perfect-mvp", "Proxy onRestoreInstanceState
Presenter = " + mPresenter);
        mBundle = savedInstanceState;
    }
}

```

6.最后V层实现，首先实现设置工厂的接口，然后创建一个代理并传入默认工厂，在V层生命周期中使用代理去实现管理Presenter的生命周期

```

/**
 * @author 刘稼旗
 * @date 2017/11/17
 * @description 继承自Activity的基类MvpActivity
 * 使用代理模式来代理Presenter的创建、销毁、绑定、解绑以及
Presenter的状态保存,其实就是管理Presenter的生命周期
 */
public abstract class AbstractMvpActivitiy<V extends
BaseMvpview, P extends BaseMvpPresenter<V>> extends
Activity implements PresenterProxyInterface<V,P> {
    private static final String PRESENTER_SAVE_KEY =
"presenter_save_key";
    /**
     * 创建被代理对象,传入默认Presenter的工厂
     */
    private BaseMvpProxy<V,P> mProxy = new BaseMvpProxy<>(
PresenterMvpFactoryImpl.<V,P>createFactory(getClass()));
    @Override
    protected void onCreate(@Nullable Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.e("perfect-mvp", "V onCreate");
        Log.e("perfect-mvp", "V onCreate mProxy = " +
mProxy);
        Log.e("perfect-mvp", "V onCreate this = " +
this.hashCode());
        if(savedInstanceState != null){

```

```
mProxy.onRestoreInstanceState(savedInstanceState.getBundle(PRESENTER_SAVE_KEY));
    }
}
@Override
protected void onResume() {
    super.onResume();
    Log.e("perfect-mvp","V onResume");
    mProxy.onResume((V) this);
}
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.e("perfect-mvp","V onDestroy = " );
    mProxy.onDestroy();
}
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.e("perfect-mvp","V onSaveInstanceState");

    outState.putBundle(PRESENTER_SAVE_KEY,mProxy.onSaveInstanceState());
}
@Override
public void
setPresenterFactory(PresenterMvpFactory<V, P>
presenterFactory) {
    Log.e("perfect-mvp","V setPresenterFactory");
    mProxy.setPresenterFactory(presenterFactory);
}
@Override
public PresenterMvpFactory<V, P>
getPresenterFactory() {
    Log.e("perfect-mvp","V getPresenterFactory");
    return mProxy.getPresenterFactory();
}
```

```
@Override  
public P getMvpPresenter() {  
    Log.e("perfect-mvp", "V getMvpPresenter");  
    return mProxy.getMvpPresenter();  
}  
}
```

最后看一下使用，首先在V层上定义需要创建的Presenter，声明自己模块具体的View接口类型和Presenter类型，最后实现自己View接口就ok了，还有就是如果要设置自己的Presenter创建工厂，必须在调用onResume方法和getMvpPresenter方法之前设置

```
//声明需要创建的Presenter  
@CreatePresenter(RequestPresenter5.class)  
public class MainActivity extends  
AbstractMvpAppCompatActivity<RequestView5,  
RequestPresenter5> implements RequestView5 {  
    @Fieldview(R.id.tv_text)  
    private TextView textView;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        viewFind.bind(this);  
        //设置自己的Presenter工厂，如果你想自定义的话  
        // setPresenterFactory(xxx);  
        if(savedInstanceState != null){  
            Log.e("perfect-mvp", "MainActivity  onCreate 测试 = " + savedInstanceState.getString("test"));  
        }  
    }  
    //点击事件  
    public void request(View view) {  
        Log.e("perfect-mvp", "点击事件");  
        getMvpPresenter().clickRequest("101010100");  
    }  
    @Override  
    public void requestLoading() {
```

```
        textView.setText("请求中，请稍后...");  
    }  
    @Override  
    protected void onSaveInstanceState(Bundle outState) {  
        super.onSaveInstanceState(outState);  
        Log.e("perfect-mvp", "MainActivity  
onSaveInstanceState 测试");  
        outState.putString("test", "test_save");  
    }  
    @Override  
    public void resultSuccess(WeatherBean result) {  
        //成功  
  
        textView.setText(result.getWeatherInfo().toString());  
    }  
    @Override  
    public void resultFailure(String result) {  
        //失败  
        textView.setText(result);  
    }  
}
```

这时候如果界面意外销毁,Presenter可以通过重写以下方法进行释放资源, 存储数据, 和恢复数据, 例如:

```
@Override  
public void onCreatePersenter(@Nullable Bundle  
savedState) {  
    super.onCreatePersenter(savedState);  
    if(savedState != null){  
        Log.e("perfect-mvp", "RequestPresenter5  
onCreatePersenter 测试 = " +  
        savedState.getString("test2") );  
    }  
}  
@Override  
public void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);
```

```
    Log.e("perfect-mvp", "RequestPresenter5  
onSaveInstanceState 测试 " );  
    outState.putString("test2", "test_save2");  
}  
@Override  
public void onDestroyPersenter() {  
    super.onDestroyPersenter();  
}
```

哦了！大功告成，perfect！可能有的人会说这只是Activity，那么Fragment中怎么弄呢，其实是一模一样的，我们将实现全部抽离到了代理中，那么Fragment中也只需要创建一个代理，然后在生命周期中使用代理调用相应就好。

当然最后我的库中已经实现了Fragment的基类和AppCompatActivity的基类，至于View如果有使用到的可以自行扩展，再次声明本文只是提供一种思路和封装的方法，并不代表就是最好的，如果有更好的想法和思路可以一起探讨。

最后奉上github地址，还希望可以给个star鼓励一下。再次感谢！

GitHub地址：

<https://github.com/ljqloveyou123>

1.1.7 Android---我所理解的MVP模式

前言

两年前第一次接触MVP模式，就被各种接口各种分层给弄的云里雾里。相信大多数的朋友第一次接触MVP的实例就是网上泛滥的那个登录模型，没错，就是这个没有任何卵用的模型让我认识到最初的MVP模式。随着这两年频繁的接触MVP，对这种质壁分离的模式也有了些小小的见解，下面就来分析一下。

模型简介

MVP模式简单的理解可以概括为将一个业务拆分成Model，View，Presenter三层结构。

Model层主管数据处理，包括网络数据，数据库数据，文件IO读写等；

View层主管UI更新和与UI相关的简单逻辑；

Presenter就是连接Model层和View层之间的桥梁，沟通Model和View进行各种交流，其作用一方面是获取Model层数据，二就是通过拿到的数据，进行部分逻辑处理后，交由View层进行UI更新。

这篇文章需要有一定的MVP基础，不然你可能会懵逼

乞丐版MVP

我这里将未经封装的MVP模式简称为乞丐版MVP，这种MVP只是简单的使用了MVP的思想，粗暴的写出来M，V，P三层结构，未经封装，存在大量的代码冗余和部分内存泄漏的风险，所以乞丐版MVP只是用来理解MVP模式，不能直接放在项目中使用。

简单模拟一个场景：假设在一个Activity中有一个网络请求，拿到网络数据后在P层进行相关逻辑处理后，在V层进行相应数据更新。

我们从M层先写，一步一步倒追到V层，首先M层有一个网络请求：

```
public class DemoModel {  
    public void onRequestData(final Callback<DemoBean>  
        callback) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //模拟网络请求  
                SystemClock.sleep(5000);  
                //网络请求成功  
                callback.onSuccess(new DemoBean("哈哈"));  
            }  
        }).start();  
    }  
}
```

根据MVP的思想，V层和M层不能有直接的交流，那么肯定就是P层来进行调用M层的代码：

```
public class DemoPresenter {  
  
    private IDemoView mView;  
    private DemoModel mModel;  
  
    public DemoPresenter(IDemoView view) {  
        mView = view;  
        mModel = new DemoModel();  
    }  
  
    public void onRequestData() {  
        mModel.onRequestData(new Callback<DemoBean>() {  
            @Override  
            public void onSuccess(DemoBean data) {  
                mView.updateUI(data);  
            }  
        });  
    }  
}
```

V层和P层直接是通过接口的形式IDemoView进行交互，当然P层和M层也可以使用接口交互：

```
public interface IDemoView {  
  
    void updateUI(DemoBean data);  
}
```

V层也就是相当于我们的Activity：

```
public class DemoActivity extends AppCompatActivity
implements IDemoView{

    private DemoPresenter mPresenter = new
DemoPresenter(this);

    @Override
    protected void onCreate(@Nullable Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        mPresenter.onRequestData();
    }

    @Override
    public void updateUI(DemoBean data) {
        Toast.makeText(this, data.str,
Toast.LENGTH_SHORT).show();
    }
}
```

这就是一个最基本的MVP结构，V层和M层阻隔，P作为中枢系统，将M层拿到的数据通过一定的逻辑处理后，交由V层进行UI更新。

乞丐版MVP有以下几个缺点：

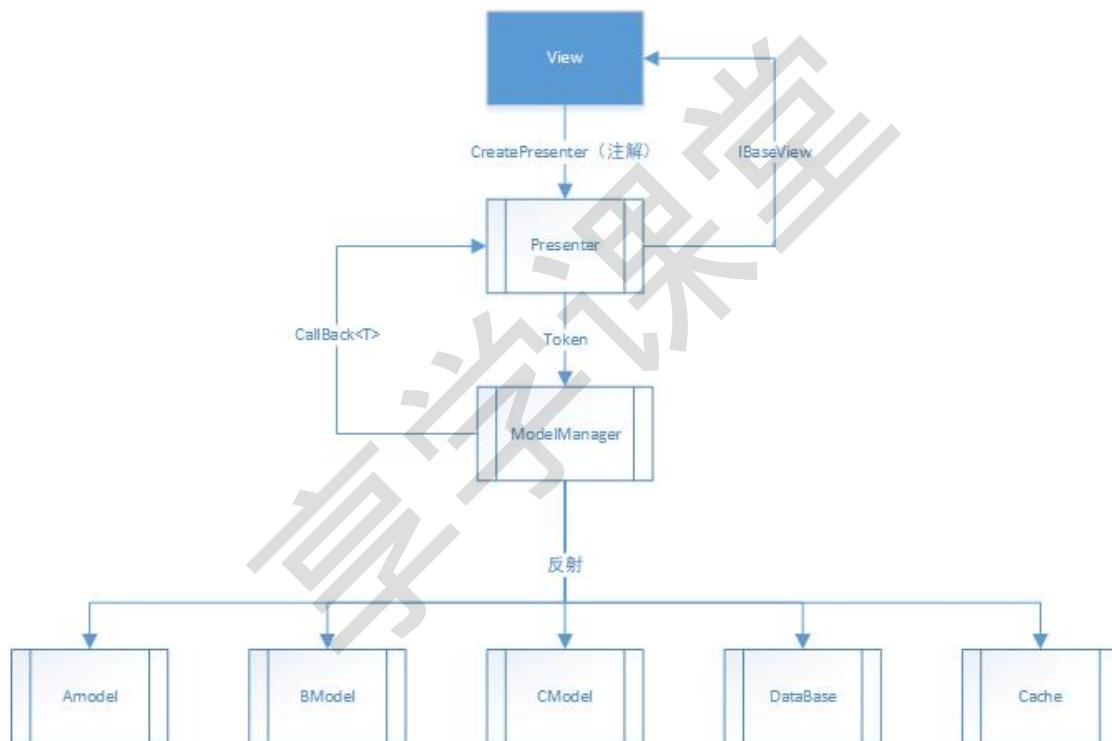
- 1.V中使用P都需要new一个出来，不同的V使用的P不同，但创建形式相同，代码存在冗余；
- 2.P中使用M也是在构造器中new一个M，M作为网络请求，可能会存在多种场景复用的情况，这种写法阻碍了代码的复用；
- 3.V层被干掉，P层却没有收到相应关闭的通知，存在一定的内存泄漏；

封装后的MVP框架

针对乞丐版MVP，这里我做了以下的几点封装：

1. 创建V, P, M层对应的基类，如：BaseMVPActivity, BasePresenter, BaseModel；
2. 使用注解的形式，在BaseMVPActivity中自动创建出我们所需要的P，减少代码冗余；
3. 创建一个ModelManager管理类，使用反射的形式创建出Model，减少P和M之间的耦合，增强复用性；
4. P层中增加其生命周期方法，绑定于对应的V层，V层被干掉的情况下，保证P也会结束掉相应的工作，减少内存泄漏；
5. 使用Loader自动管理P，防止在屏幕状态改变的情况下，创建多个P。

先看一个结构图：



MVP框架流程图.jpg

结构图上可以清晰的看到调用情况，V层通过注解的形式创建P，P通过IBaseView通知V，P通过Token类加载ModelManager获取对应的M对象，ModelManager通过CallBack回调来通知P拿取数据，ModelManager使用反射的形式管理众多M，这样就成功的将M层单独抽离成一个大模块。

废话不多说，首先先看下使用情况：

V层：

```
@CreatePresenter(MainPresenter.class)
public class MainActivity extends
BaseMVPActivity<MainPresenter> implements IMainview {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onPresenterCreate() {
        mPresenter.show();
    }

    @Override
    public void show() {
        Log.d("MainActivity", "呵呵");
    }
}
```

P层：

```
public class MainPresenter extends
BasePresenter<IMainView> {

    public void show() {
        HttpParam param = HttpParam.obtain();
        param.put(APIParamKey.PHONE, "13112345678");

        ModelManager.getModel(Token.MAIN_MODE)
            .setParam(param)
            .execute(new CallBack<MainBean>() {
                @Override
```

```
        public void onSuccess(MainBean data)
    {
        mview.show();
    }

    @Override
    public void onFailure(String msg) {

    }
}

}
```

M层：

```
public class MainModel extends BaseModel<MainBean> {

    @Override
    public void execute(final callBack<MainBean>
callBack) {
        super.execute(callBack);
        HttpParam param = getParam();
        String phone = param.get(APIParamKey.PHONE);
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                callBack.onSuccess(new MainBean());
            }
        }, 4000);
    }
}
```

封装后的结构比之前要清晰许多，代码也少了一部分，M层和P层也实现了完全的解耦合。

那么我们来先看看M层和P层是怎么样实现完全的解耦合的呢：

M层主要封装：

ModelManager:

```
public class ModelManager {

    private static ArrayMap<String, IBaseModel> mCache =
new ArrayMap<>();

    /**
     * 传递类，反射获得该类对象
     *
     * @param token 类引用字符串
     */
    public static IBaseModel getModel(Class<? extends BaseModel> token) {
        return getModel(token.getName());
    }

    /**
     * 传递类引用字符串，反射获得该类对象
     *
     * @param token 类引用字符串
     */
    public static IBaseModel getModel(String token) {
        IBaseModel model = mCache.get(token);
        try {
            if (model == null) {
                model = (IBaseModel)
Class.forName(token).newInstance();
                mCache.put(token, model);
            }
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
        return model;
    }
}
```

主要是使用反射来获取对应的Model类，增加mCache缓存实现容器类单例。这里我实现了两个方法，一个是传递Class类，一个是传递字符串。我个人建议是使用传递字符串的方式来创建Model，通过一个类Token，封装所有的Model信息，使P和M直接只通过ModelManager的形式来交互，也便于M的复用。

ModelManager第二个方法是传递一个类的引用字符串，我把所有要引用的字符串写到一个类里，便于管理，起名叫Token：

```
public interface Token {
    //    String MAIN_MODE = "com.zdu.mvpdemo.MainModel";
    String MAIN_MODE = MainModel.class.getName();
}
```

我写了两种方式来引用字符串，推荐使用第二种，使用XXXModel.class.getName()的方式来获取当前类的引用。让Token和各个Model直接建立起联系，便于查找，同时也能防止混淆时Model类因为没有任何引用而被认为是无效类从而被删掉的问题。

BaseModel类中主要实现了4个方法：

参数的set和get方法：

```
@Override
public BaseModel<T> setParam(HttpParam param) {
    this.mParam = param;
    mParam.setUse(true);
    return this;
}
```

```
@Override
public HttpParam getParam() {
```

```
    if (mParam == null) {
        throw new NullPointerException("入参为空");
    }
    return mParam;
}
```

HttpParam类是我封装的一个ArrayMap集合，内部加入了缓冲池的概念。优点就是可以重复利用，缺点就是无法动态的指定Map集合的大小。至于怎么实现的缓存池，请看我写的另一篇文章：[Android 模拟 Message.obtain\(\)，构建自己的缓存池。](#)

两个执行方法 execute(CallBack)，和空参execute()

```
/**
 * 执行异步操作
 *
 * @param callback<T> 回调
 */
@Override
public void execute(@Nullable Callback<T> callback) {
    recycleParam();
}

/**
 * 执行异步操作，无需回调
 */
@Override
public void execute() {
    execute(null);
}

/**
 * 回收
 */
private void recycleParam() {
    if (mParam != null) {
        mParam.setUse(false);
```

```
        mParam.recycle();
    }
}
```

两个方法我都没设置成抽象的，简单实现了下回收的方法，一个带回调，一个不带回调。子类想调用哪个就实现哪个方法就可以了

V层主要封装：

V层主要实现BaseMVPActivity，封装了以下几个功能：

- 基类自动与P层进行关联，并进行生命周期绑定
- 注解@CreatePresenter的形式自动创建P
- 使用Loader管理P的加载，防止屏幕状态改变时，创建多个P

BaseMVPActivity：

```
public class BaseMVPActivity<P extends BasePresenter>
extends BaseActivity implements
LoaderManager.LoaderCallbacks<P>, IBaseview {

    private final int LOADER_ID = TAG.hashCode();
    protected P mPresenter;

    @Override
    protected void onCreate(@Nullable Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportLoaderManager().initLoader(LOADER_ID,
savedInstanceState, this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        if (mPresenter != null) {
            mPresenter.onPause();
        }
    }
}
```

```
}
```

```
@Override  
protected void onResume() {  
    super.onResume();  
    if (mPresenter != null) {  
        mPresenter.onResume();  
    }  
}
```

```
@Override  
protected void onDestroy() {  
    super.onDestroy();  
    if (mPresenter != null) {  
        mPresenter.onDetach();  
    }  
    mPresenter = null;  
}
```

```
@NonNull
```

```
@Override  
public Loader<P> onCreateLoader(int id, @Nullable  
Bundle args) {  
    return new PresenterLoader<>(this, getClass());  
}
```

```
@Override  
public void onLoadFinished(@NonNull Loader<P> loader,  
P data) {  
    if (data != null) {  
        mPresenter = data;  
        mPresenter.onAttach(this);  
        onPresenterCreate();  
    }  
}
```

```
@Override
```

```
public void onLoaderReset(@NonNull Loader<P> loader)
{
    if (mPresenter != null) {
        mPresenter.onDetach();
    }
    mPresenter = null;
}

/**
 * 回调此方法，presenter创建完毕
 */
protected void onPresenterCreate() {

}

}
```

首先在onCreate方法初始化Loader，在实现的onCreateLoader方法里加载注解解析器PresenterLoader，加载完成实现方法onLoadFinished进行P的赋值和接口的绑定，数据被重置时实现onLoaderReset，接触对P的绑定，对应的生命周期方法里也将P绑定在一起。

注解解析器PresenterLoader：

```
public class PresenterLoader<P extends BasePresenter>
extends Loader<P> {

    private CreatePresenter mCreatePresenter;
    private P mPresenter;

    public PresenterLoader(Context context, Class<?>
tClass) {
        super(context);
        mCreatePresenter =
tClass.getAnnotation(CreatePresenter.class);
    }
}
```

```

private P getPresenter() {
    if (mCreatePresenter != null) {
        Class<P> pClass = (Class<P>)
mCreatePresenter.value();
        try {
            return pClass.newInstance();
        } catch (Exception e) {
            throw new RuntimeException("Presenter创建
失败!, 检查是否声明了@CreatePresenter(xx.class)注解");
        }
    }
    return null;
}

@Override
protected void onStartLoading() {
    super.onStartLoading();
    if (mPresenter == null) {
        forceLoad();
    } else {
        deliverResult(mPresenter);
    }
}

@Override
protected void onForceLoad() {
    super.onForceLoad();
    mPresenter = getPresenter();
    deliverResult(mPresenter);
}

```

这个类的作用其一就是绑定Loader，管理P的创建加载情况，其二就是获取注解对象，通过反射来创建P。

注解类就很简单了：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface CreatePresenter {
    Class<? extends BasePresenter> value();
}
```

有一定的约束条件，必须继承BasePresenter才能通过注解来创建。

P层的封装：

V和M都搞定了，作为桥梁的P层就简单了：

BasePresenter：

```
public class BasePresenter<V extends IBaseview>
implements IBasePresenter<V> {

    protected V mview;

    @Override
    public void onAttach(V view) {
        mview = view;
    }

    @Override
    public V getView() {
        checkViewAttached();
        return mview;
    }

    @Override
    public void onPause() {
    }

    @Override
    public void onResume() {
    }
}
```

```
@Override
public boolean isViewAttached() {
    return mview != null;
}

@Override
public void checkViewAttached() {
    if (!isViewAttached()) {
        throw new NullPointerException("view 已为空");
    }
}

/**
 * 取消，置空数据，防止内存泄露
 */
@Override
public void onDetach() {
    mview = null;
}

}
```

自动装箱：onAttach(), 自动拆箱：onDetach(), 还有各种生命周期，逻辑处理起来就方便了很多。

综述

通过一定的封装，创建P的过程交给基类通过注解的形式来创建，减少了代码的冗余；给P一定的生命周期方法，在V被干掉的情况下，P能及时处理对应的事件，减少内存泄漏；通过ModelManager统一管理Model，使用Token类作为Model和ModelManager的桥梁，彻底的将P和M进行解耦，同时分离了M层，使M层抽出成一个独立的模块，增加了M的复用性。

附上Demo的GitHub地址：[MVPDemo](#)。

第二节 组件化架构

业务大了代码多了会用到。

为什么要用组件化？

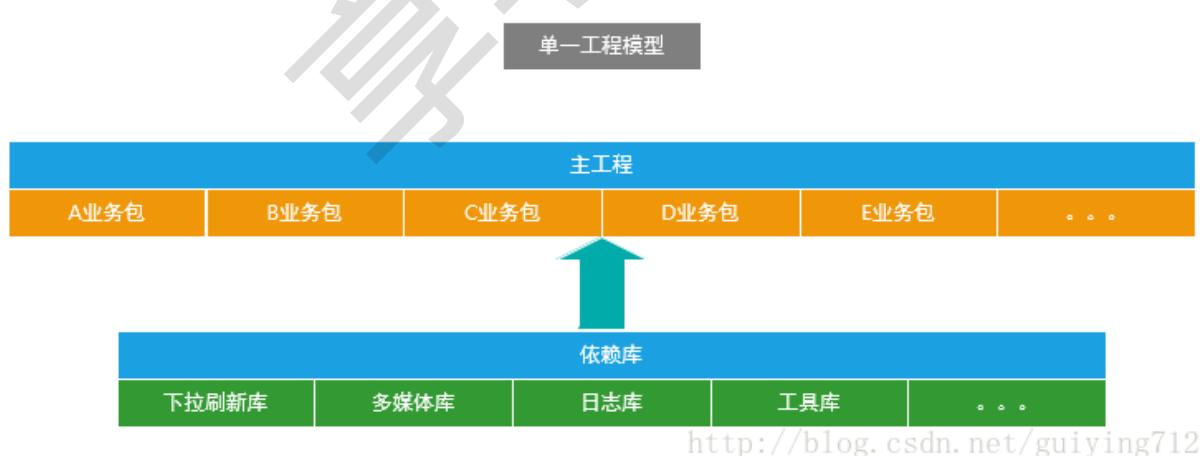
组件之间如何通信？

组件之间如何跳转？

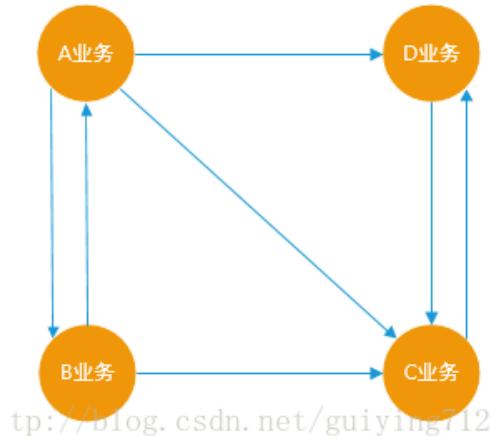
2.1 Android为什么要用组件化？

2.1.1 为什么要项目组件化

随着APP版本不断的迭代，新功能的不断增加，业务也会变的越来越复杂，APP业务模块的数量有可能还会继续增加，而且每个模块的代码也变的越来越多，这样发展下去单一工程下的APP架构势必会影响开发效率，增加项目的维护成本，每个工程师都要熟悉如此之多的代码，将很难进行多人协作开发，而且Android项目在编译代码的时候电脑会非常卡，又因为单一工程下代码耦合严重，每修改一处代码后都要重新编译打包测试，导致非常耗时，最重要的是这样的代码想要做单元测试根本无从下手，所以必须要有更灵活的架构代替过去单一的工程架构。



上图是目前比较普遍使用的Android APP技术架构，往往是在一个界面中存在大量的业务逻辑，而业务逻辑中充斥着各种网络请求、数据操作等行为，整个项目中也没有模块的概念，只有简单的以业务逻辑划分的文件夹，并且业务之间也是直接相互调用、高度耦合在一起的；



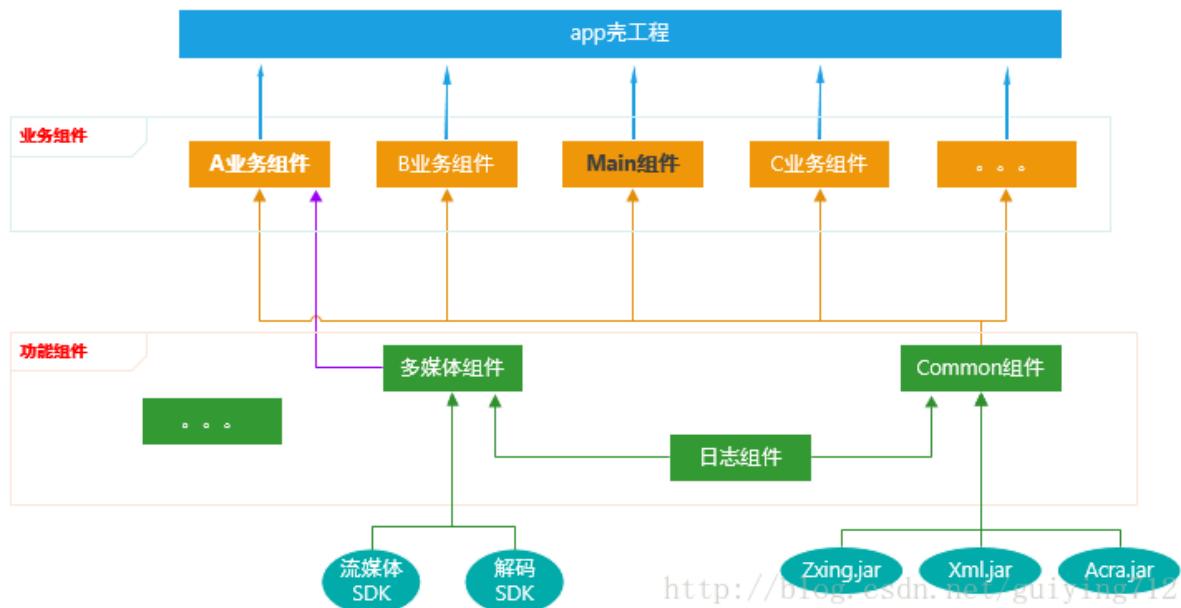
上图单一工程模型下的业务关系，总的来说就是：你中有我，我中有你，相互依赖，无法分离。

然而随着产品的迭代，业务越来越复杂，随之带来的是项目结构复杂度的极度增加，此时我们会面临如下几个问题：

- 1、实际业务变化非常快，但是单一工程的业务模块耦合度太高，牵一发而动全身；
- 2、对工程所做的任何修改都必须要编译整个工程；
- 3、功能测试和系统测试每次都要进行；
- 4、团队协同开发存在较多的冲突.不得不花费更多的时间去沟通和协调，并且在开发过程中，任何一位成员没办法专注于自己的功能点，影响开发效率；
- 5、不能灵活的对业务模块进行配置和组装；

为了满足各个业务模块的迭代而彼此不受影响，更好的解决上面这种让人头疼的依赖关系，就需要整改App的架构。

2.1.2如何组件化



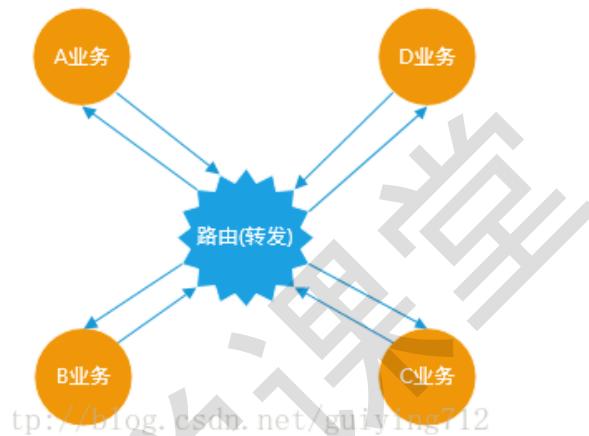
上图是组件化工程模型，为了方便理解这张架构图，下面会列举一些组件化工程中用到的名词的含义：

| 名词 | 含义 |
|----------|--|
| 集成模式 | 所有的业务组件被“app壳工程”依赖，组成一个完整的APP； |
| 组件模式 | 可以独立开发业务组件，每一个业务组件就是一个APP； |
| app壳工程 | 负责管理各个业务组件，和打包apk，没有具体的业务功能； |
| 业务组件 | 根据公司具体业务而独立形成一个的工程； |
| 功能组件 | 提供开发APP的某些基础功能，例如打印日志、树状图等； |
| Main组件 | 属于业务组件，指定APP启动页面、主界面； |
| Common组件 | 属于功能组件，支撑业务组件的基础，提供多数业务组件需要的功能，例如提供网络请求功能； |

**

Android APP组件化架构的目标是告别结构臃肿，让各个业务变得相对独立，业务组件在组件模式下可以独立开发，而在集成模式下又可以变为aar包集成到“app壳工程”中，组成一个完整功能的APP；

从组件化工程模型中可以看到，**业务组件之间是独立的，没有关联的**，这些业务组件在集成模式下是一个个library，被app壳工程所依赖，组成一个具有完整业务功能的APP应用，但是在组件开发模式下，业务组件又变成了一个个application，它们可以独立开发和调试，由于在组件开发模式下，业务组件们的代码量相比于完整的项目差了很远，因此在运行时可以显著减少编译时间。



这是组件化工程模型下的业务关系，业务之间将不再直接引用和依赖，而是通过“路由”这样一个中转站间接产生联系，而Android中的路由实际就是对URL Scheme的封装；

如此规模大的架构整改需要付出更高的成本，还会涉及一些潜在的风险，但是整改后的架构能够带来很多好处：

- 1、加快业务迭代速度，各个业务模块组件更加独立，不再出现业务耦合情况；
- 2、稳定的公共模块采用依赖库方式，提供给各个业务线使用，减少重复开发和维护工作量；
- 3、迭代频繁的业务模块采用组件方式，各业务研发可以互不干扰、提升协作效率，并控制产品质量；
- 4、为新业务随时集成提供了基础，所有业务可上可下，灵活多变；
- 5、降低团队成员熟悉项目的成本，降低项目的维护难度；
- 6、加快编译速度，提高开发效率；
- 7、控制代码权限，将代码的权限细分到更小的粒度；

2.1.3组件化实施流程

1) 组件模式和集成模式的转换

Android Studio中的Module主要有两种属性，分别为：

1、**application属性**，可以独立运行的Android程序，也就是我们的APP；

```
apply plugin: 'com.android.application'
```

2、**library属性**，不可以独立运行，一般是Android程序依赖的库文件；

```
apply plugin: 'com.android.library'
```

Module的属性是在每个组件的 **build.gradle** 文件中配置的，当我们在组件模式开发时，业务组件应处于application属性，这时的业务组件就是一个 Android App，可以独立开发和调试；而当我们转换到集成模式开发时，业务组件应该处于 library 属性，这样才能被我们的“app壳工程”所依赖，组成一个具有完整功能的APP；

但是我们如何让组件在这两种模式之间自动转换呢？总不能每次需要转换模式的时候去每个业务组件的 Gralde 文件中去手动把 Application 改成 library 吧？如果我们的项目只有两三个组件那么这个办法肯定是可行的，手动去改一遍也用不了多久，但是在大型项目中我们可能会有十几个业务组件，再去手动改一遍必定费时费力，这时候就需要程序员发挥下懒的本质了。

试想，我们经常在写代码的时候定义静态常量，那么定义静态常量的目的什么呢？当一个常量需要被好几处代码引用的时候，把这个常量定义为静态常量的好处是当这个常量的值需要改变时我们只需要改变静态常量的值，其他引用了这个静态常量的地方都会被改变，**做到了一次改变，到处生效**；根据这个思想，那么我们就可以在我们的代码中的某处定义一个决定业务组件属性的常量，然后让所有业务组件的build.gradle都引用这个常量，这样当我们改变了常量值的时候，所有引用了这个常量值的业务组件就会根据值的变化改变自己的属性；可是问题来了？静态常量是用Java代码定义的，而改变组件属性是需要在Gradle中定义的，Gradle能做到吗？

Gradle自动构建工具有一个重要属性，可以帮助我们完成这个事情。每当我们用AndroidStudio创建一个Android项目后，就会在项目的根目录中生成一个文件 **gradle.properties**，我们将使用这个文件的一个重要属性：**在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来**；那么我们在上面提到解决办法就有了实际行动的方法，首先我们在gradle.properties中定义一个常量值 **isModule**（是否是组件开发模式，**true**为是，**false**为否）：

```
# 每次更改“isModule”的值后，需要点击 "Sync Project" 按钮  
isModule=false
```

然后我们在业务组件的build.gradle中读取 **isModule**，但是gradle.properties还有一个重要属性：**gradle.properties 中的数据类型都是String类型，使用其他数据类型需要自行转换**；也就是说我们读到isModule 是个String类型的值，而我们需要的是Boolean值，代码如下：

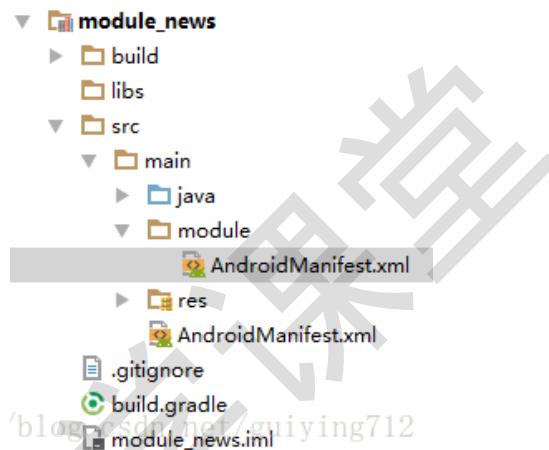
```
if (isModule.toBoolean()) {  
    apply plugin: 'com.android.application'  
} else {  
    apply plugin: 'com.android.library'  
}
```

这样我们第一个问题就解决了，当然了 **每次改变isModule的值后，都要同步项目才能生效**；

2) 组件之间AndroidManifest合并问题

在 AndroidStudio 中每一个组件都会有对应的 AndroidManifest.xml，用于声明需要的权限、Application、Activity、Service、Broadcast等，当项目处于组件模式时，业务组件的 AndroidManifest.xml 应该具有一个 Android APP 所具有的的所有属性，尤其是声明 Application 和要 launch 的 Activity，但是当项目处于集成模式的时候，每一个业务组件的 AndroidManifest.xml 都要合并到“app壳工程”中，要是每一个业务组件都有自己的 Application 和 launch 的 Activity，那么合并的时候肯定会冲突，试想一个APP怎么可能会有多个 Application 和 launch 的 Activity 呢？

但是大家应该注意到这个问题是在组件开发模式和集成开发模式之间转换引起的问题，而在上一节中我们已经解决了组件模式和集成模式转换的问题，另外大家应该都经历过将 Android 项目从 Eclipse 切换到 AndroidStudio 的过程，由于 Android 项目在 Eclipse 和 AndroidStudio 开发时 AndroidManifest.xml 文件的位置是不一样的，我们需要在 **build.gradle** 中指定下 **AndroidManifest.xml** 的位置，AndroidStudio 才能读取到 **AndroidManifest.xml**，这样解决办法也就有了，我们可以为 **组件开发模式下的业务组件再创建一个 **AndroidManifest.xml****，然后根据 **isModule** 指定 **AndroidManifest.xml** 的文件路径，让业务组件在集成模式和组件模式下使用不同的 **AndroidManifest.xml**，这样表单冲突的问题就可以规避了。



上图是组件化项目中一个标准的业务组件目录结构，首先我们在 main 文件夹下创建一个 module 文件夹用于存放组件开发模式下业务组件的 **AndroidManifest.xml**，而 **AndroidStudio** 生成的 **AndroidManifest.xml** 则依然保留，并用于集成开发模式下业务组件的表单；然后我们需要在业务组件的 **build.gradle** 中指定表单的路径，代码如下：

```
sourceSets {  
    main {  
        if (isModule.toBoolean()) {  
            manifest.srcFile  
'src/main/module/AndroidManifest.xml'  
        } else {  
            manifest.srcFile  
'src/main/AndroidManifest.xml'  
        }  
    }  
}
```

这样在不同的开发模式下就会读取到不同的 AndroidManifest.xml，然后我们需要修改这两个表单的内容以为我们不同的开发模式服务。

首先是集成开发模式下的 AndroidManifest.xml，前面我们说过集成模式下，业务组件的表单是绝对不能拥有自己的 Application 和 launch 的 Activity的，也不能声明APP名称、图标等属性，总之app壳工程有的属性，业务组件都不能有，下面是一份**标准的集成开发模式**下业务组件的 AndroidManifest.xml：

```
<manifest  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
    package="com.guiying.girls">  
  
    <application android:theme="@style/AppTheme">  
        <activity  
            android:name=".main.GirlsActivity"  
            android:screenOrientation="portrait" />  
        <activity  
            android:name=".girl.GirlActivity"  
            android:screenOrientation="portrait"  
            android:theme="@style/AppTheme.NoActionBar"  
        />  
        </application>  
  
</manifest>
```

我在这个表单中只声明了应用的主题，而且这个主题还是跟app壳工程中的主题是一致的，都引用了common组件中的资源文件，在这里声明主题是为了方便这个业务组件中有使用默认主题的Activity时就不用再给Activity单独声明theme了。

然后是组件开发模式下的表单文件：

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.guiping.girls">

    <application
        android:name="debug.GirlsApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/girls_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".main.GirlsActivity"
            android:screenOrientation="portrait">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".girl.GirlActivity"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar"
        />
    </application>
```

```
</manifest>
```

组件模式下的业务组件表单就是一个Android项目普通的AndroidManifest.xml，这里就不在过多介绍了。

3) 全局Context的获取及组件数据初始化

当Android程序启动时，Android系统会为每个程序创建一个Application类的对象，并且只创建一个，application对象的生命周期是整个程序中最长的，它的生命周期就等于这个程序的生命周期。在默认情况下应用系统会自动生成Application对象，但是如果我们自定义了Application，那就需要在AndroidManifest.xml中声明告知系统，实例化的时候，是实例化我们自定义的，而非默认的。

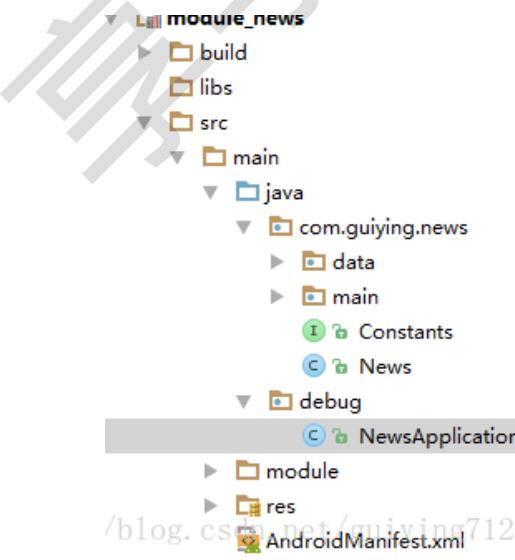
但是我们在组件化开发的时候，可能为了数据的问题每一个组件都会自定义一个Application类，如果我们在自己的组件中开发时需要获取**全局的Context**，一般都会直接获取application对象，但是当所有组件要打包合并在一起的时候就会出现问题，因为最后程序只有一个Application，我们组件中自己定义的Application肯定是没法使用的，因此我们需要想办法再任何一个业务组件中都能获取到全局的Context，而且这个Context不管是在组件开发模式还是在集成开发模式都是生效的。

在组件化工程模型图中，功能组件集合中有一个**Common组件**，Common有公共、公用、共同的意思，所以这个组件中主要封装了项目中需要的基础功能，并且每一个业务组件都要依赖Common组件，Common组件就像是万丈高楼的地基，而业务组件就是在Common组件这个地基上搭建起来我们的APP的，Common组件会专门在一个章节中讲解，这里只讲Common组件中的一个功能，在Common组件中我们封装了项目中用到的各种Base类，这些基类中就有**BaseApplication类**。

BaseApplication主要用于各个业务组件和app壳工程中声明的Application类继承用的，只要各个业务组件和app壳工程中声明的Application类继承了BaseApplication，当应用启动时BaseApplication就会被自动实例化，这样从BaseApplication获取的Context就会生效，也就从根本上解决了我们不能直接从各个组件获取全局Context的问题；

这时候大家肯定都会有个疑问？不是说了业务组件不能有自己的 Application 吗，怎么还让他们继承 BaseApplication 呢？其实我前面说的是业务组件不能在集成模式下拥有自己的 Application，但是这不代表业务组件也不能在组件开发模式下拥有自己的 Application，其实业务组件在组件开发模式下必须要有自己的 Application 类，一方面是为了让 BaseApplication 被实例化从而获取 Context，还有一个作用是，**业务组件自己的 Application 可以在组件开发模式下初始化一些数据**，例如在组件开发模式下，A组件没有登录页面也没法登录，因此就无法获取到 Token，这样请求网络就无法成功，因此我们需要在A组件这个 APP 启动后就应该已经登录了，这时候组件自己的 Application 类就有了用武之地，我们在组件的 Application 的 onCreate 方法中模拟一个登陆接口，在登陆成功后将数据保存到本地，这样就可以处理A组件中的数据业务了；另外我们也可以在组件Application中初始化一些第三方库。

但是，实际上业务组件中的Application在最终的集成项目中是没有什么实际作用的，组件自己的 Application 仅限于在组件模式下发挥功能，因此我们需要在将项目从组件模式转换到集成模式后将组件自己的Application剔除出我们的项目；在 AndroidManifest 合并问题小节中介绍了如何在不同开发模式下让 Gradle 识别组件表单的路径，这个方法也同样适用于 Java 代码；



我们在Java文件夹下创建一个 debug 文件夹，用于存放不会在业务组件中引用的类，例如上图中的 NewsApplication，你甚至可以在 **debug 文件夹中创建一个Activity，然后组件表单中声明启动这个Activity，在这个Activity中不用setContentView，只需要在启动你的目标Activity的**

时候传递参数就行，这样就可以解决组件模式下某些Activity需要 getIntent数据而没有办法拿到的情况，代码如下；

```
public class LauncherActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        request();
        Intent intent = new Intent(this,
TargetActivity.class);
        intent.putExtra("name", "avcd");
        intent.putExtra("syscode", "023e2e12ed");
        startActivity(intent);
        finish();
    }

    //申请读写权限
    private void request() {
        AndPermission.with(this)
            .requestCode(110)

            .permission(Manifest.permission.WRITE_EXTERNAL_STORAGE,
                Manifest.permission.CAMERA,
                Manifest.permission.READ_PHONE_STATE)
            .callback(this)
            .start();
    }

}
```

接下来在业务组件的 build.gradle 中，根据 isModule 是否是集成模式将 debug 这个 Java 代码文件夹排除：

```
sourceSets {  
    main {  
        if (isModule.toBoolean()) {  
            manifest.srcFile  
'src/main/module/AndroidManifest.xml'  
        } else {  
            manifest.srcFile  
'src/main/AndroidManifest.xml'  
            //集成开发模式下排除debug文件夹中的所有Java文件  
            java {  
                exclude 'debug/**'  
            }  
        }  
    }  
}
```

4) library依赖问题

在介绍这一节的时候，先说一个问题，在**组件化工程模型图**中，多媒体组件和Common组件都依赖了日志组件，而A业务组件有同时依赖了多媒体组件和Common组件，这时候就会有人问，你这样搞岂不是日志组件要被重复依赖了，而且Common组件也被每一个业务组件依赖了，这样不出问题吗？

其实大家完全没有必要担心这个问题，如果真有重复依赖的问题，在你编译打包的时候就会报错，如果你还是不相信的话可以反编译下最后打包出来的APP，看看里面的代码你就知道了。组件只是我们在代码开发阶段中为了方便叫的一个术语，在组件被打包进APP的时候是没有这个概念的，这些组件最后都会被打包成arr包，然后被app壳工程所依赖，在构建APP的过程中Gradle会自动将重复的arr包排除，APP中也就不会存在相同的代码了；

但是虽然组件是不会重复了，但是我们还是要考虑另一个情况，我们在build.gradle中compile的第三方库，例如AndroidSupport库经常会被一些开源的控件所依赖，而我们自己一定也会compile AndroidSupport库，这就会造成第三方包和我们自己的包存在重复加载，解决办法就是找出那个多出来的库，并将多出来的库给排除掉，而且Gradle也是支持这样做

的，分别有两种方式：根据组件名排除或者根据包名排除，下面以排除 support-v4 库为例：

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
  
    compile("com.jude:easyrecyclerview:$rootProject.easyRecycleVersion") {  
        exclude module: 'support-v4'//根据组件名排除  
        exclude group: 'android.support.v4'//根据包名排除  
    }  
}
```

library 重复依赖的问题算是都解决了，但是我们在开发项目的时候会依赖很多开源库，而这些库每个组件都需要用到，要是每个组件都去依赖一遍也是很麻烦的，尤其是给这些库升级的时候，为了方便我们统一管理第三方库，我们将给整个工程提供统一的依赖第三方库的入口，前面介绍的 Common 库的作用之一就是统一依赖开源库，因为其他业务组件都依赖了 Common 库，所以这些业务组件也就间接依赖了 Common 所依赖的开源库。

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    //Android Support  
    compile "com.android.support:appcompat-v7:$rootProject.supportLibraryVersion"  
    compile  
    "com.android.support:design:$rootProject.supportLibraryVersion"  
    compile  
    "com.android.support:percent:$rootProject.supportLibraryVersion"  
    //网络请求相关  
    compile  
    "com.squareup.retrofit2:retrofit:$rootProject.RetrofitVersion"  
    compile "com.squareup.retrofit2:retrofit-mock:$rootProject.RetrofitVersion"
```

```
compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieVersion"
    //稳定的
    compile
"com.github.bumptech.glide:glide:$rootProject.glideversion"
    compile
"com.orhanobut:logger:$rootProject.loggerVersion"
    compile
"org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile
"com.google.code.gson:gson:$rootProject.gsonVersion"
    compile
"com.github.chrisbanes:PhotoView:$rootProject.photoViewVersion"

    compile
"com.jude:easyrecyclerview:$rootProject.easyRecyclerview"
    compile
"com.github.GrenderG:Toasty:$rootProject.toastyVersion"

    //router
    compile
"com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

5) 组件之间调用和通信

在组件化开发的时候，组件之间是没有依赖关系，我们不能在使用显示调用来跳转页面了，因为我们组件化的目的之一就是解决模块间的强依赖问题，假如现在要从A业务组件跳转到业务B组件，并且要携带参数跳转，这时候怎么办呢？而且组件这么多怎么管理也是个问题，这时候就需要引入“路由”的概念了，由本文开始的组件化模型下的业务关系图可知路由就是起到一个转发的作用。

这里我将介绍开源库的“**ActivityRouter**”，有兴趣的同学请直接去ActivityRouter的Github主页学习：[ActivityRouter](#)，ActivityRouter支持给Activity定义 URL，这样就可以通过 URL 跳转到Activity，并且支持从浏览器以及 APP 中跳入我们的Activity，而且还支持通过 url 调用方法。下面将介绍如何将ActivityRouter集成到组件化项目中以实现组件之间的调用；

1、首先我们需要在 Common 组件中的 build.gradle 将ActivityRouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //router
    compile
    "com.github.mzule.activityrouter:activityrouter:$rootProject.routerVersion"
}
```

2、这一步我们需要先了解 **APT**这个概念，**APT(Annotation Processing Tool)**是一种处理注解的工具，它对源代码文件进行检测找出其中的**Annotation**，使用**Annotation**进行额外的处理。**Annotation处理器**在处理**Annotation**时可以根据源文件中的**Annotation**生成额外的源文件和其它的文件(文件具体内容由**Annotation处理器**的编写者决定)，**APT**还会编译生成的源文件和原来的源文件，将它们一起生成**class**文件。在这里我们将在每一个**业务组件**的 build.gradle 都引入ActivityRouter 的 Annotation处理器，我们将会在声明组件和Url的时候使用，annotationProcessor是Android官方提供的Annotation处理器插件，代码如下：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
    "com.github.mzule.activityrouter:compiler:$rootProject.annotationProcessor"
}
```

3、接下来需要在 **app壳工程** 的 `AndroidManifest.xml` 配置，到这里 `ActivityRouter` 配置就算完成了：

```
<!-- 声明整个应用程序的路由协议-->
<activity
    android:name="com.github.mzule.activityrouter.router.RouterActivity"
    android:theme="@android:style/Theme.NoDisplay">
    <intent-filter>
        <action
            android:name="android.intent.action.VIEW" />
        <category
            android:name="android.intent.category.DEFAULT" />
        <category
            android:name="android.intent.category.BROWSABLE" />
        <data
            android:scheme="@string/global_scheme" /> <!-- 改成自己的
            scheme -->
    </intent-filter>
</activity>
<!-- 发送崩溃日志界面-->
```

4、接下来我们将声明项目中的业务组件，声明方法如下：

```
@Module("girls")
public class Girls { }
```

在每一个业务组件的 `java` 文件的根目录下创建一个类，用 **注解@Module** 声明这个业务组件；

然后在“**app壳工程**”的 **应用Application** 中使用 **注解@Module** 管理我们声明的所有业务组件，方法如下：

```
@Modules({"main", "girls", "news"})
public class MyApplication extends BaseApplication {
}
```

到这里组件化项目中的所有业务组件就被声明和管理起来了，组件之间的也就可以互相调用了，当然前提是给业务组件中的Activity定义URL。

5、例如我们给 Girls组件 中的 GirlsActivity 使用注解@Router 定义一个URL：“news”，方法如下：

```
@Router("girls")
public class GirlsActivity extends BaseActionBarActivity {
    private Girlsview mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new Girlsview(this);
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}
```

然后我们就可以在项目中的任何一个地方通过 URL地址：
module://girls, 调用 GirlsActivity，方法如下：

```
Routers.open(MainActivity.this, "module://girls");
```

组件之间的调用解决后，另外需要解决的就是组件之间的通信，例如A业务组件中有消息列表，而用户在B组件中操作某个事件后会产生一条新消息，需要通知A组件刷新消息列表，这样业务场景需求可以使用Android广播来解决，也可以使用第三方的事件总线来实现，比如EventBus。

6) 组件之间资源名冲突

因为我们拆分出了很多业务组件和功能组件，在把这些组件合并到“**app壳工程**”时候就有可能会出现资源名冲突问题，例如**A组件和B组件都有一张叫做“ic_back”的图标**，这时候在集成模式下打包APP就会编译出错，解决这个问题最简单的办法就是在**项目中约定资源文件命名规约**，比如强制使每个资源文件的名称以组件名开始，这个可以根据实际情况和开发人员制定规则。当然了万能的Gradle构建工具也提供了解决方法，通过在在组件的build.gradle中添加如下的代码：

```
//设置了resourcePrefix值后，所有的资源名必须以指定的字符串做前缀，否则会报错。  
//但是resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。  
resourcePrefix "girls_"
```

但是设置了这个属性后有个问题，所有的资源名必须以指定的字符串做前缀，否则会报错，而且resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名；所以我并不推荐使用这种方法来解决资源名冲突。

2.1.4组件化项目的工程类型

在组件化工程模型中主要有：**app壳工程、业务组件和功能组件3种类型，而业务组件中的Main组件和功能组件中的Common组件比较特殊**，下面将分别介绍。

1) app壳工程

app壳工程是从名称来解释就是一个空壳工程，没有任何的业务代码，也不能有Activity，但它又必须被单独划分成一个组件，而不能融合到其他组件中，是因为它有如下几点重要功能：

1、app壳工程中声明了我们Android应用的 Application，这个 Application 必须继承自 Common 组件中的 BaseApplication（如果你无需实现自己的 Application 可以直接在表单声明 BaseApplication），因为只有这样，在打包应用后才能让 BaseApplication 中的 Context 生效，当然你还可以在这个 Application 中初始化我们工程中使用到的库文件，还可以在这里解决 Android 引用方法数不能超过 65535 的限制，对崩溃事件的捕获和发送也可以在这里声明。

2、app壳工程的 AndroidManifest.xml 是我Android应用的根表单，应用的名称、图标以及是否支持备份等等属性都是在这份表单中配置的，其他组件中的表单最终在集成开发模式下都被合并到这份 AndroidManifest.xml 中。

3、app壳工程的 build.gradle 是比较特殊的，app壳不管是在集成开发模式还是组件开发模式，它的属性始终都是： com.android.application，因为最终其他的组件都要被 app 壳工程所依赖，被打包进 app 壳工程中，这一点从组件化工程模型图中就能体现出来，所以 app 壳工程是不需要单独调试单独开发的。另外 Android 应用的打包签名，以及 buildTypes 和 defaultConfig 都需要在这里配置，而它的 dependencies 则需要根据 isModule 的值分别依赖不同的组件，在组件开发模式下 app 壳工程只需要依赖 Common 组件，或者为了防止报错也可以根据实际情况依赖其他功能组件，而在集成模式下 app 壳工程必须依赖所有在应用 Application 中声明的业务组件，并且不需要再依赖任何功能组件。

下面是一份 app 壳工程的 build.gradle 文件：

```
apply plugin: 'com.android.application'

static def buildTime() {
    return new Date().format("yyyyMMdd");
}

android {
    signingConfigs {
        release {
            keyAlias 'guiying712'
```

```
        keyPassword 'guiying712'
        storeFile file('/mykey.jks')
        storePassword 'guiying712'
    }

}

compileSdkVersion rootProject.ext.compileSdkVersion
buildToolsVersion rootProject.ext.buildToolsVersion
defaultConfig {
    applicationId "com.guiying.androidmodulepattern"
    minSdkVersion rootProject.ext.minSdkVersion
    targetSdkVersion rootProject.ext.targetSdkVersion
    versionCode rootProject.ext.versionCode
    versionName rootProject.ext.versionName
    multiDexEnabled false
    //打包时间
    resValue "string", "build_time", buildTime()
}

buildTypes {
    release {
        //更改AndroidManifest.xml中预先定义好占位符信息
        //manifestPlaceholders = [app_icon:
        "@drawable/icon"]
        // 不显示Log
        buildConfigField "boolean", "LEO_DEBUG",
        "false"
        //是否zip对齐
        zipAlignEnabled true
        // 缩减resource文件
        shrinkResources true
        //Proguard
        minifyEnabled true
        proguardFiles
        getDefaultProguardFile('proguard-android.txt'),
        'proguard-rules.pro'
        //签名
        signingConfig signingConfigs.release
    }
}
```

```
    }

    debug {
        //给applicationId添加后缀“.debug”
        applicationIdSuffix ".debug"
        //manifestPlaceholders = [app_icon:
        "@drawable/launch_beta"]
        buildConfigField "boolean", "LOG_DEBUG",
        "true"
        zipAlignEnabled false
        shrinkResources false
        minifyEnabled false
        debuggable true
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    annotationProcessor
    "com.github.mzule.activityrouter:compiler:$rootProject.an
    notationProcessor"
    if (isModule.toBoolean()) {
        compile project(':lib_common')
    } else {
        compile project(':module_main')
        compile project(':module_girls')
        compile project(':module_news')
    }
}
```

2) 功能组件和Common组件

功能组件是为了支撑业务组件的某些功能而独立划分出来的组件，功能实质上跟项目中引入的第三方库是一样的，功能组件的特征如下：

- 1、功能组件的 AndroidManifest.xml 是一张空表，这张表中只有功能组件的包名；
- 2、功能组件不管是在集成开发模式下还是组件开发模式下属性始终是：com.android.library，所以功能组件是不需要读取 gradle.properties 中的 isModule 值的；另外功能组件的 build.gradle 也无需设置 buildTypes，只需要 dependencies 这个功能组件需要的jar包和开源库。

下面是一份普通的功能组件的 build.gradle文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Common组件除了有功能组件的普遍属性外，还具有其他功能：

- 1、Common组件的 AndroidManifest.xml 不是一张空表，这张表中声明了我们 Android应用用到的所有使用权限 uses-permission 和 uses-feature，放到这里是因为在组件开发模式下，所有业务组件就无需在自己的 AndroidManifest.xml 声明自己要用到的权限了。
- 2、Common组件的 build.gradle 需要统一依赖业务组件中用到的第三方依赖库和jar包，例如我们用到的ActivityRouter、Okhttp等等。

3、Common组件中封装了Android应用的 Base类和网络请求工具、图片加载工具等等，公用的 widget控件也应该放在Common 组件中；业务组件中都用到的数据也应放于Common组件中，例如保存到 SharedPreferences 和 DataBase 中的登陆数据；

4、Common组件的资源文件中需要放置项目公用的 Drawable、layout、string、dimen、color和style 等等，另外项目中的 Activity 主题必须定义在 Common中，方便和 BaseActivity 配合保持整个Android应用的界面风格统一。

下面是一份 Common功能组件的 build.gradle文件：

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion

    defaultConfig {
        minSdkVersion rootProject.ext.minSdkVersion
        targetSdkVersion rootProject.ext.targetSdkVersion
        versionCode rootProject.ext.versionCode
        versionName rootProject.ext.versionName
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    //Android Support
    compile "com.android.support:appcompat-
v7:$rootProject.supportLibraryVersion"
    compile
    "com.android.support:design:$rootProject.supportLibraryVe
rsion"
    compile
    "com.android.support:percent:$rootProject.supportLibraryV
ersion"
```

```
//网络请求相关
compile
"com.squareup.retrofit2:retrofit:$rootProject.retrofitVersion"
    compile "com.squareup.retrofit2:retrofit-
mock:$rootProject.retrofitVersion"
    compile
"com.github.franmontiel:PersistentCookieJar:$rootProject.cookieversion"
    //稳定的
    compile
"com.github.bumptech.glide:glide:$rootProject.glideversion"
    compile
"com.orhanobut:logger:$rootProject.loggerVersion"
    compile
"org.greenrobot:eventbus:$rootProject.eventbusVersion"
    compile
"com.google.code.gson:gson:$rootProject.gsonversion"
    compile
"com.github.chrisbanes:PhotoView:$rootProject.photoviewVe-
rsion"
    compile
"com.jude:easyrecyclerview:$rootProject.easyRecyclerversi-
on"
    compile
"com.github.GrenderG:Toasty:$rootProject.toastyVersion"

//router
compile
"com.github.mzule.activityrouter:activityrouter:$rootProj-
ect.routerVersion"
}
```

3) 业务组件和Main组件

业务组件就是根据业务逻辑的不同拆分出来的组件，业务组件的特征如下：

- 1、业务组件中要有两张AndroidManifest.xml，分别对应组件开发模式和集成开发模式，这两张表的区别请查看 **组件之间AndroidManifest合并问题** 小节。
- 2、业务组件在集成模式下是不能有自己的Application的，但在组件开发模式下又必须实现自己的Application并且要继承自Common组件的BaseApplication，并且这个Application不能被业务组件中的代码引用，因为它的功能就是为了使业务组件从BaseApplication中获取的全局Context生效，还有初始化数据之用。
- 3、业务组件有debug文件夹，这个文件夹在集成模式下会从业务组件的代码中排除掉，所以debug文件夹中的类不能被业务组件强引用，例如组件模式下的 Application 就是置于这个文件夹中，还有组件模式下开发给目标 Activity 传递参数的用的 launch Activity 也应该置于 debug 文件夹中；
- 4、业务组件必须在自己的 Java文件夹中创建业务组件声明类，以使 **app壳工程 中的应用Application能够引用**，实现组件跳转，具体请查看 **组件之间调用和通信** 小节；
- 5、**业务组件必须在自己的 build.gradle 中根据 isModule 值的不同改变自己的属性**，在组件模式下是：com.android.application，而在集成模式下com.android.library；同时还需要在build.gradle配置资源文件，如指定不同开发模式下的AndroidManifest.xml文件路径，排除debug文件夹等；业务组件还必须在dependencies中依赖Common组件，并且引入ActivityRouter的注解处理器annotationProcessor，以及依赖其他用到的功能组件。

下面是一份普通业务组件的 build.gradle文件：

```
if (isModule.toBoolean()) {  
    apply plugin: 'com.android.application'  
} else {  
    apply plugin: 'com.android.library'  
}
```

```
        android {  
            compileSdkVersion rootProject.ext.compileSdkVersion  
            buildToolsVersion rootProject.ext.buildToolsVersion  
  
            defaultConfig {  
                minSdkVersion rootProject.ext.minSdkVersion  
                targetSdkVersion rootProject.ext.targetSdkVersion  
                versionCode rootProject.ext.versionCode  
                versionName rootProject.ext.versionName  
            }  
  
            sourceSets {  
                main {  
                    if (isModule.toBoolean()) {  
                        manifest.srcFile  
                        'src/main/module/AndroidManifest.xml'  
                    } else {  
                        manifest.srcFile  
                        'src/main/AndroidManifest.xml'  
                    }  
                    //集成开发模式下排除debug文件夹中的所有Java文件  
                    java {  
                        exclude 'debug/**'  
                    }  
                }  
            }  
  
            //设置了resourcePrefix值后，所有的资源名必须以指定的字符串做  
            //前缀，否则会报错。  
            //但是resourcePrefix这个值只能限定xml里面的资源，并不能限定  
            //图片资源，所有图片资源仍然需要手动去修改资源名。  
            //resourcePrefix "girls_"  
        }  
  
        dependencies {  
            compile fileTree(dir: 'libs', include: ['*.jar'])
```

```
annotationProcessor
"com.github.mzule.activityrouter:compiler:$rootProject.an
notationProcessor"
    compile project(':lib_common')
}
```

Main组件除了有业务组件的普遍属性外，还有一项重要功能：

1、Main组件集成模式下的AndroidManifest.xml是跟其他业务组件不一样的，Main组件的表单中声明了我们整个Android应用的launch Activity，这就是Main组件的独特之处；所以我建议SplashActivity、登陆Activity以及主界面都应属于Main组件，也就是说Android应用启动后要调用的页面应置于Main组件。

```
<activity
    android:name=".splash.SplashActivity"
    android:launchMode="singleTop"
    android:screenOrientation="portrait"
    android:theme="@style/SplashTheme">
    <intent-filter>
        <action
            android:name="android.intent.action.MAIN" />
        <category
            android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

2.1.5组件化项目的混淆方案

组件化项目的Java代码混淆方案采用在集成模式下集中在app壳工程中混淆，各个业务组件不配置混淆文件。集成开发模式下在app壳工程中build.gradle文件的release构建类型中开启混淆属性，其他buildTypes配置方案跟普通项目保持一致，Java混淆配置文件也放置在app壳工程中，各个业务组件的混淆配置规则都应该在app壳工程中的混淆配置文件中添加和修改。

之所以不采用在每个业务组件中开启混淆的方案，是因为 **组件在集成模式下都被 Gradle 构建成了 release 类型的jar包**，一旦业务组件的代码被混淆，而这时候代码中又出现了bug，将很难根据日志找出导致bug的原因；另外每个业务组件中都保留一份混淆配置文件非常不便于修改和管理，这也是不推荐在业务组件的 build.gradle 文件中配置 buildTypes（构建类型）的原因。

2.1.6 工程的build.gradle和gradle.properties文件

1) 组件化工程的build.gradle文件

在组件化项目中因为每个组件的 build.gradle 都需要配置 compileSdkVersion、buildToolsVersion 和 defaultConfig 等的版本号，而且每个组件都需要用到 annotationProcessor，**为了能够使组件化项目中的所有组件的 build.gradle 中的这些配置都能保持统一，并且也是为了方便修改版本号**，我们统一在Android工程根目录下的build.gradle中定义这些版本号，当然为了方便管理Common组件中的第三方开源库的版本号，最好也在这里定义这些开源库的版本号，然后在各个组件的 build.gradle 中引用Android工程根目录下的build.gradle定义的版本号，组件化工程的 build.gradle 文件代码如下：

```
buildscript {  
    repositories {  
        jcenter()  
        mavenCentral()  
    }  
  
    dependencies {  
        //classpath  
        "com.android.tools.build:gradle:$localGradlePluginVersion"  
        "  
            // $localGradlePluginVersion 是 gradle.properties 中的数据  
            classpath  
        "com.android.tools.build:gradle:$localGradlePluginVersion"  
        "  
    }  
}
```

```
}

allprojects {
    repositories {
        jcenter()
        mavenCentral()
        //Add the JitPack repository
        maven { url "https://jitpack.io" }
        //支持aar包
        flatDir {
            dirs 'libs'
        }
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

// Define versions in a single place
//时间: 2017.2.13; 每次修改版本号都要添加修改时间
ext {
    // Sdk and tools
    //localBuildToolsVersion是gradle.properties中的数据
    buildToolsVersion = localBuildToolsVersion
    compileSdkVersion = 25
    minSdkVersion = 16
    targetSdkVersion = 25
    versionCode = 1
    versionName = "1.0"
    javaVersion = JavaVersion.VERSION_1_8

    // App dependencies version
    supportLibraryVersion = "25.3.1"
    retrofitVersion = "2.1.0"
    glideVersion = "3.7.0"
    loggerVersion = "1.15"
    eventbusVersion = "3.0.0"
```

```
        gsonVersion = "2.8.0"
        photoviewVersion = "2.0.0"

        //需检查升级版本
        annotationProcessor = "1.1.7"
        routerVersion = "1.2.2"
        easyRecyclerVersion = "4.4.0"
        cookieVersion = "v1.0.1"
        toastifyVersion = "1.1.3"
    }
```

2) 组件化工程的gradle.properties文件

在组件化实施流程中我们了解到gradle.properties有两个属性对我们非常有用：

- 1、在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来，不管这个build.gradle是组件的还是整个项目工程的build.gradle；
- 2、gradle.properties中的数据类型都是String类型，使用其他数据类型需要自行转换；

利用gradle.properties的属性不仅可以解决集成开发模式和组件开发模式的转换，而且还可以解决在多人协同开发Android项目的时候，因为开发团队成员的Android开发环境（开发环境指Android SDK和AndroidStudio）不一致而导致频繁改变线上项目的build.gradle配置。

在每个Android组件的 build.gradle 中有一个属性：

buildToolsVersion，表示构建工具的版本号，这个属性值对应AndroidSDK 中的 **Android SDK Build-tools**，正常情况下 build.gradle 中的 buildToolsVersion 跟你电脑中 Android SDK Build-tools 的最新版本是一致的，比如现在 Android SDK Build-tools 的最新的版本是：25.0.3，那么我的Android项目中 build.gradle 中的 buildToolsVersion 版本号也是 25.0.3，但是一旦一个Android项目是由好几个人同时开发，总会出现每个人的开发环境 Android SDK Build-tools 是都是不一样的，并不是所有人都会经常升级更新 Android SDK，而且代码是保存到线上环

境的（例如使用 SVN/Git 等工具），某个开发人员提交代码后线上 Android 项目中 build.gradle 中的 buildToolsVersion 也会被不断地改变。

另外一个原因是因为 Android 工程的根目录下的 build.gradle 声明了 Android Gradle 构建工具，而这个工具也是有版本号的，而且 **Gradle Build Tools** 的版本号跟 Android Studio 版本号一致的，但是有些开发人员基本很久都不会升级自己的 Android Studio 版本，导致团队中每个开发人员的 Gradle Build Tools 的版本号也不一致。

如果每次同步代码后这两个工具的版本号被改变了，开发人员可以自己手动改回来，并且不要把改动工具版本号的代码提交到线上环境，这样还可以勉强继续开发；但是很多公司都会使用持续集成工具（例如 Jenkins）用于持续的软件版本发布，而 Android 出包是需要 Android SDK Build-tools 和 Gradle Build Tools 配合的，一旦提交到线上的版本跟持续集成工具所依赖的 Android 环境构建工具版本号不一致就会导致 Android 打包失败。

为了解决上面问题就必须将 Android 项目中 build.gradle 中的 buildToolsVersion 和 GradleBuildTools 版本号从线上代码隔离出来，保证线上代码的 buildToolsVersion 和 Gradle Build Tools 版本号不会被人为改变。

7、组件化项目 Router 的其他方案 - ARouter

在组件化项目中使用到的跨组件跳转库 ActivityRouter 可以使用阿里巴巴的开源路由项目：[阿里巴巴ARouter](#)；

ActivityRouter 和 ARouter 的接入组件化项目的方式是一样的，ActivityRouter 提供的功能目前 ARouter 也全部支持，但是 ARouter 还支持依赖注入解耦，页面、拦截器、服务等组件均会自动注册到框架。对于大家来说，没有最好的只有最适合的，大家可以根据自己的项目选择合适的 Router。

下面将介绍 ARouter 的基础使用方法，更多功能还需大家去 Github 自己学习；

1、首先 ARouter 这个框架是需要初始化SDK的，所以你需要在“app壳工程”中的应用Application中加入下面的代码，**注意：在 debug 模式下一定要 openDebug**：

```
if (BuildConfig.DEBUG) {  
    //一定要在ARouter.init之前调用openDebug  
    ARouter.openDebug();  
    ARouter.openLog();  
}  
ARouter.init(this);
```

2、首先我们依然需要在 Common 组件中的 build.gradle 将ARouter 依赖进来，方便我们在业务组件中调用：

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    //router  
    compile 'com.alibaba:arouter-api:1.2.1.1'  
}
```

3、然后在每一个**业务组件**的 build.gradle 都引入ARouter 的 Annotation 处理器，代码如下：

```
android {  
    defaultConfig {  
        ...  
        javaCompileOptions {  
            annotationProcessorOptions {  
                arguments = [ moduleName : project.getName() ]  
            }  
        }  
    }  
}  
  
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    annotationProcessor 'com.alibaba:arouter-compiler:1.0.3'
```

```
}
```

4、由于ARouter支持自动注册到框架，所以我们不用像ActivityRouter那样在各个组件中声明组件，当然更不需要在Application中管理组件了。我们给 Girls组件 中的 GirlsActivity 添加注解：@Route(path = "/girls/list")，需要注意的是这里的路径至少需要有两级，/xx/xx，之所以这样是因为**ARouter使用了路径中第一段字符串(/*)作为分组**，比如像上面的“girls”，而分组这个概念就有点类似于ActivityRouter中的组件声明@Module，代码如下：

```
@Route(path = "/girls/list")
public class GirlsActivity extends BaseActionBarActivity {
    private GirlsView mView;
    private GirlsContract.Presenter mPresenter;

    @Override
    protected int setTitleId() {
        return R.string.girls_activity_title;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mView = new GirlsView(this);
        setContentView(mView);
        mPresenter = new GirlsPresenter(mView);
        mPresenter.start();
    }
}
```

然后我们就可以在项目中的任何一个地方通过 URL地址：/girls/list，调用 GirlsActivity，方法如下：

```
ARouter.getInstance().build("/girls/list").navigation();
```

2.1.7 结束语

组件化相比于单一工程优势是显而易见的：

1. 组件模式下可以加快编译速度，提高开发效率；
2. 自由选择开发框架（MVC / MVP / MVVM /）；
3. 方便做单元测试；
4. 代码架构更加清晰，降低项目的维护难度；
5. 适合于团队开发；

最后贴出Android组件化Demo地址：[Android组件化项目](#)
[AndroidModulePattern](#)

第六章 性能优化篇

【面试重点】性能优化：包括启动优化（主要是冷启动）、内存优化、绘制优化、稳定性优化、安装包体积优化等，优化是面试的重中之重。

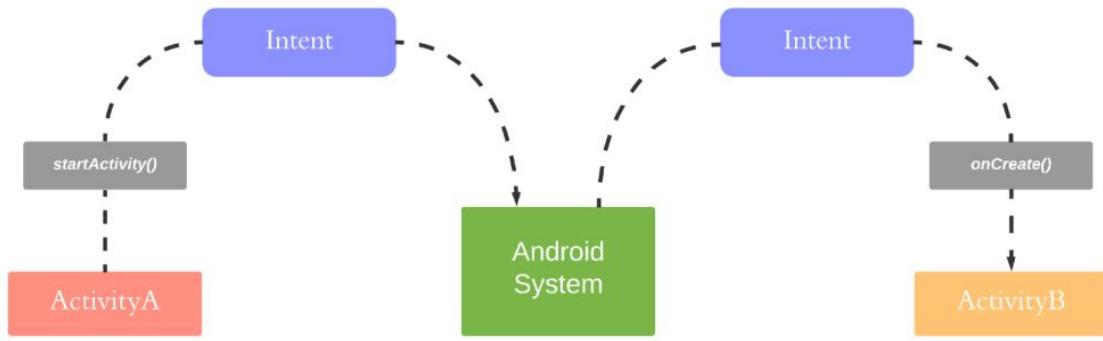
你在开发中是如何做性能优化的？

第一节 启动优化

1.1 Activity启动流程

本文分析的源代码基于 android27

Activity的启动流程相当复杂，比如我们在Activity A 打开Activity B**。这一过程开始于A.startActivity 经过 Android System 的处理，最终调用 B** 的生命周期方法结束。



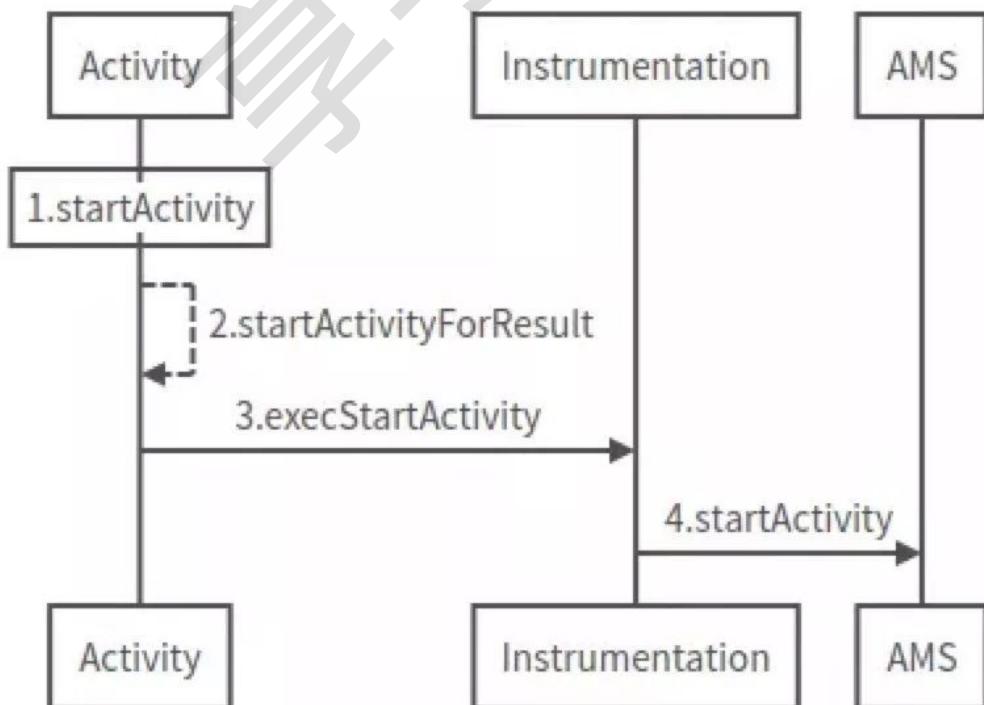
QQ 算法详解

上面的过程具体描述的话太过于复杂，我们可以将其拆解成3部分依次突破。

1. startActivity --> ActivityManagerService
2. ActivityManagerService --> ApplicationThread
3. ApplicationThread --> Activity

(一) startActivity --> ActivityManagerService

ActivityManagerService后续简称 AMS。先放一张时序图：



startActivity→AMS时序图.png

QQ 算法详解

从Activity的startActivity方法到AMS中间的过程并不复杂，下面看下源码中做了哪些操作：

Activity的startActivity

```
@Override  
public void startActivityForResult(Intent intent, @Nullable Bundle  
options) {  
    if (options != null) {  
        //第二个参数为-1，表示不需要知道Activity启动的结果  
        startActivityForResult(intent, -1, options);  
    } else {  
        startActivityForResult(intent, -1);  
    }  
}
```

startActivity中调用startActivityForResult

Activity的startActivityForResult

```
public void startActivityForResult(@RequiresPermission  
Intent intent, int requestCode, @Nullable Bundle options)  
{  
    if (mParent == null) { //mParent表示当前Activity的  
        //父类，一般为null  
        options =  
        transferSpringboardActivityOptions(options);  
  
        //调用Instrumentation.execStartActivity()，启  
动新的Activity。  
        //mMainThread类型为ActivityThread，在attach()  
        //时被回调时被赋值。  
        Instrumentation.ActivityResult ar =  
        mInstrumentation.execStartActivity(this,  
        mMainThread.getApplicationThread(), mToken, this, intent,  
        requestCode, options);  
        if (ar != null) {  
            mMainThread.sendActivityResult(
```

```
mToken, mEmbeddedID, requestCode,  
ar.getResultCode(),  
        ar.getResultData());  
    }  
    if (requestCode >= 0) {  
        mStartedActivity = true;  
    }  
  
    cancelInputsAndStartExitTransition(options);  
} else {  
    ...  
}  
}
```

startActivityForResult也很简单，调用
Instrumentation.execStartActivity方法。剩下的交给Instrumentation类
去处理。

1. **Instrumentation**类主要用来监控应用程序与系统交互
2. 红色字体标明的mMainThread是**ActivityThread**类型，
ActivityThread可以理解为一个进程，在这就是A所在的进程
3. 通过mMainThread获取一个**ApplicationThread**的引用，这个引用
就是用来实现进程间通信的，具体来说就 AMS 所在的系统进程
通知应用程序进程进行一系列操作，后面会用到

Instrumentation的execStartActivity

```
public ActivityResult execStartActivity(  
        Context who, IBinder contextThread, IBinder  
token, Activity target,  
        Intent intent, int requestCode, Bundle  
options) {  
  
    //将contextThread转成ApplicationThread.  
    IApplicationThread whoThread =  
(IApplicationThread) contextThread;  
    /*
```

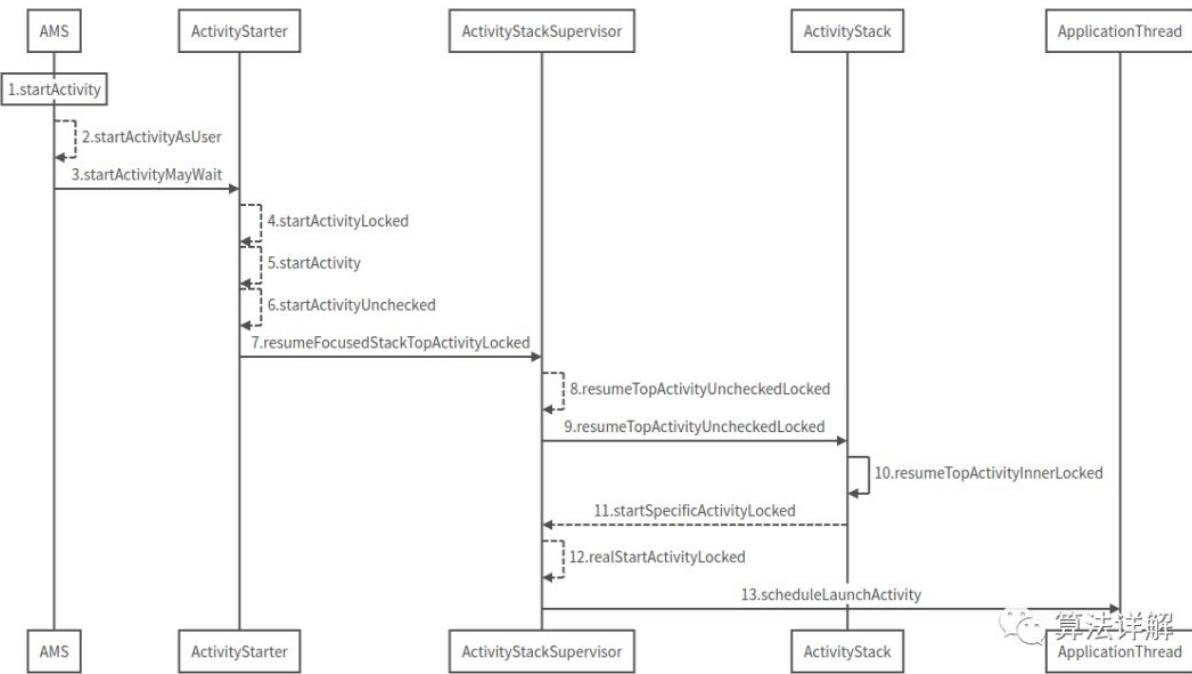
```
* 中间代码省略
*/
try {
    intent.migrateExtraStreamToClipData();
    intent.prepareToLeaveProcess(who);
    //实际上这里是通过AIDL来调用AMS的startActivity方法，下面我们看下ActivityManager.getService()的代码
    int result = ActivityManager.getService()
        .startActivity(whoThread,
    who.getPackageName(), intent,
    intent.resolveTypeIfNeeded(who.getContentResolver()),
    token, target != null ?
    target.mEmbeddedID : null,
    requestCode, 0, null, options);
    checkStartActivityResult(result, intent); //检查StartActivity的结果
} catch (RemoteException e) {
    throw new RuntimeException("Failure from system", e);
}
return null;
}
```

熟悉Android Binder机制的同学应该很快就能定位到此处实际调用到了ActivityManagerService的startActivity方法。接下来的工作重心就由**A**进程转移到了AMS所在的系统进程。

(二) AMS --> ApplicationThread

接下来就看下在AMS中是如何一步一步执行到**B**进程的。这里先剧透一下：刚才在看**Instrumentation**的时候，我们讲过一个**ApplicationThread**类，这个类是实现进程间通信的，这里**AMS**最终其实也就是调用了**B**进程中一个**ApplicationThread**引用，从而间接的通知**B**进程进行相应操作。

还是线上时序图：



相比于startActivity-->AMS， AMS-->ApplicationThread流程看起来复杂好多了， 实际上这里面就干了2件事

- 1 综合处理launchMode和Intent中的Flag标志位，并根据处理结果生成一个目标Activity **B**的对象(**ActivityRecord**)
- 2 判断是否需要为目标Activity **B**创建一个新的进程(**ProcessRecord**)、新的任务栈(**TaskRecord**)
- 3 来来回回的操作AMS或者ActivityStackSupervisor中的那几个比较核心的数据结构

接下来就从AMS的startActivity方法开始看起：

AMS的startActivity

```
@Override
    public final int startActivity(IApplicationThread
caller, String callingPackage,
        Intent intent, String resolvedType, IBinder
resultTo, String resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo,
Bundle bOptions) {
    return startActivityAsUser(caller,
callingPackage, intent, resolvedType, resultTo,
        resultWho, requestCode, startFlags,
profilerInfo, bOptions,
        UserHandle.getCallingUserId());
}
```

很简单就是调用了startActivityAsUser

AMS的startActivityAsUser

```
@Override
    public final int
startActivityAsUser(IApplicationThread caller, String
callingPackage,
                    Intent intent, String resolvedType, IBinder
resultTo, String resultWho, int requestCode,
                    int startFlags, ProfilerInfo profilerInfo,
Bundle bOptions, int userId) {
    enforceNotIsolatedCaller("startActivity");//检查
调用者是否合法
    userId =
mUserController.handleIncomingUser(Binder.getCallingPid()
, Binder.getCallingUid(),
userId, false, ALLOW_FULL_ONLY,
"startActivity", null);//检查调用者是否有权限执行操作

    return
mActivityStarter.startActivityMayWait(caller, -1,
callingPackage, intent,
                    resolvedType, null, null, resultTo,
resultWho, requestCode, startFlags,
                    profilerInfo, null, null, bOptions,
false, userId, null, "startActivityAsUser");
}
```

这个方法实际上就是检查当前用户是否有权限。然后就将所有的工作交给了一个叫做ActivityStarter的类中。

ActivityStarter的startActivityMayWait

ActivityStarter这个类看名字就知道它专门负责一个Activity的启动操作。它的主要作用包括解析Intent、创建ActivityRecord、如果有可能还要创建TaskRecord。其中的实现细节可以暂时跳过，总之经过一顿猛如虎的操作之后代码会被调用到startActivityUnchecked()

```

private int startActivityUnchecked(final ActivityRecord
r, ActivityRecord sourceRecord,
        IVoiceInteractionSession voiceSession,
IVoiceInteractor voiceInteractor,
        int startFlags, boolean doResume,
ActivityOptions options, TaskRecord inTask) {
    setInitialState(r, options, inTask, doResume,
startFlags, sourceRecord, voiceSession,
        voiceInteractor); //1
    computeLaunchingTaskFlags(); //2
    computeSourceStack();
    mIntent.setFlags(mLaunchFlags); //3

    ...
}

mSupervisor.resumeFocusedStackTopActivityLocked(mTargetStack,
mStartActivity, mOptions);
}

```

注释1处用于初始化启动Activity的各种配置，在初始化前会重置各种配置再进行配置，这些配置包括：ActivityRecord、Intent、TaskRecord和LaunchFlags（启动的FLAG）等等。注释2处的computeLaunchingTaskFlags方法用于计算出启动的FLAG，并将计算的值赋值给mLaunchFlags。在注释3处将mLaunchFlags设置给Intent，达到设定Activity的启动方式的目的。接着来查看computeLaunchingTaskFlags方法。

ActivityStarter的computeLaunchingTaskFlags

```

private void computeLaunchingTaskFlags() {
    ...
    if (mInTask == null) { //1
        if (mSourceRecord == null) { //2
            if ((mLaunchFlags &
FLAG_ACTIVITY_NEW_TASK) == 0 && mInTask == null) { //3
                Slog.w(TAG, "startActivity called
from non-Activity context; forcing " +

```

```
"Intent.FLAG_ACTIVITY_NEW_TASK for: " + mIntent);
        mLaunchFlags |=
FLAG_ACTIVITY_NEW_TASK;
    }
} else if (mSourceRecord.launchMode ==
LAUNCH_SINGLE_INSTANCE) { //4
    mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
} else if (mLaunchSingleInstance ||
mLaunchSingleTask) { //5
    mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
}
}
```

计算启动的FLAG的逻辑比较复杂，这里只截取了一小部分，注释1处的TaskRecord类型的mInTask为null时，说明Activity要加入的栈不存在。因此，这一小段代码主要解决的问题就是Activity要加入的栈不存在时如何计算出启动的FLAG。注释2处，ActivityRecord类型的mSourceRecord用于描述“初始Activity”，什么是“初始Activity”呢？比如ActivityA启动了ActivityB，ActivityA就是初始Activity。同时满足注释2和注释3的条件则需要创建一个新栈。注释4处，如果“初始Activity”所在的栈只允许有一个Activity实例，则也需要创建一个新栈。注释5处，如果Launch Mode设置了singleTask或singleInstance，则也要创建一个新栈。

在ActivityStarter -> startActivityUnchecked方法中，最终调用了mSupervisor的resumeFocusedStackTopActivityLocked方法。这个mSupervisor是ActivityStackSupervisor类型，简单来说它是ActivityStack的管理类。

ActivityStackSupervisor的 resumeFocusedStackTopActivityLocked

```
boolean resumeFocusedStackTopActivityLocked(
    ActivityStack targetStack, ActivityRecord
target, ActivityOptions targetOptions) {

    //...

    if (targetStack != null &&
isFocusedStack(targetStack)) {
        return
targetStack.resumeTopActivityUncheckedLocked(target,
targetOptions);

    //...
}
```

继续往下看

ActivityStack的resumeTopActivityUncheckedLocked

```
boolean resumeTopActivityUncheckedLocked(ActivityRecord
prev, ActivityOptions options) {
    if (mStackSupervisor.inResumeTopActivity) {
        // Don't even start recursing.
        return false;
    }

    boolean result = false;
    try {
        // Protect against recursion.
        mStackSupervisor.inResumeTopActivity = true;
        result = resumeTopActivityInnerLocked(prev,
options);
    } finally {
        mStackSupervisor.inResumeTopActivity = false;
    }

    //...
    return result;
```

```
}
```

继续调用resumeTopActivityInnerLocked方法

ActivityStack的resumeTopActivityInnerLocked

```
private boolean  
resumeTopActivityInnerLocked(ActivityRecord prev,  
ActivityOptions options) {  
  
    //...  
    if (mResumedActivity != null) {  
        if (DEBUG_STATES) Slog.d(TAG_STATES,  
            "resumeTopActivityLocked: Pausing "  
+ mResumedActivity);  
  
        //暂停当前activity, 调用Activity的onPause  
        pausing |= startPausingLocked(userLeaving,  
false, next, false);  
    }  
    //...  
  
    mStackSupervisor.startSpecificActivityLocked(next, true,  
true); //启动新activity  
    //...  
}
```

这个方法虽然是叫resume..., 但是在这之前需要先调用startPausingLocked方法将之前的A给暂停, 因此在A中调用startActivity打开B时, 实际上是先执行A的onPause方法, 在执行B的onCreate生命周期。

最后代码又回到了ActivityStackSupervisor

ActivityStackSupervisor的startSpecificActivityLocked

```
void startSpecificActivityLocked(ActivityRecord  
r, boolean andResume, boolean checkConfig) {
```

```
    ProcessRecord app =
mService.getProcessRecordLocked(r.processName,
                                r.info.applicationInfo.uid, true); //获取
ProcessRecord

    r.getStack().setLaunchTime(r);

    if (app != null && app.thread != null) {
        try {
            if
((r.info.flags&ActivityInfo.FLAG_MULTIPROCESS) == 0
                ||
!"android".equals(r.info.packageName)) {

                app.addPackage(r.info.packageName,
r.info.applicationInfo.versionCode,
mService.mProcessStats);
            }
            realStartActivityLocked(r, app,
andResume, checkConfig);
            return;
        } catch (RemoteException e) {
            Slog.w(TAG, "Exception when starting
activity "
+
r.intent.getComponent().flattenToString(), e);
        }
    }

    //创建新的ProcessRecord
    mService.startProcessLocked(r.processName,
r.info.applicationInfo, true, 0,
"activity", r.intent.getComponent(),
false, false, true);
}
```

这个方法中，判断系统Activity B进程(ProcessRecord)是否正在运行，如果是则继续执行realStartActivityLocked方法，否则会调用AMS的startProcessLocked方法创建进程。

ActivityStackSupervisor的realStartActivityLocked

```
final boolean realStartActivityLocked(ActivityRecord r,
ProcessRecord app,
    boolean andResume, boolean checkConfig)
throws RemoteException {

    //...
    app.thread.scheduleLaunchActivity(new
Intent(r.intent), r.appToken,
        System.identityHashCode(r), r.info,

    mergedConfiguration.getGlobalConfiguration(),
    mergedConfiguration.getOverrideConfiguration(), r.compat,
        r.launchedFromPackage,
    task.voiceInteractor, app.repProcState, r.icicle,
        r.persistentState, results, newIntents,
!andResume,
        mService.isNextTransitionForward(),
profilerInfo);
    //...
}
```

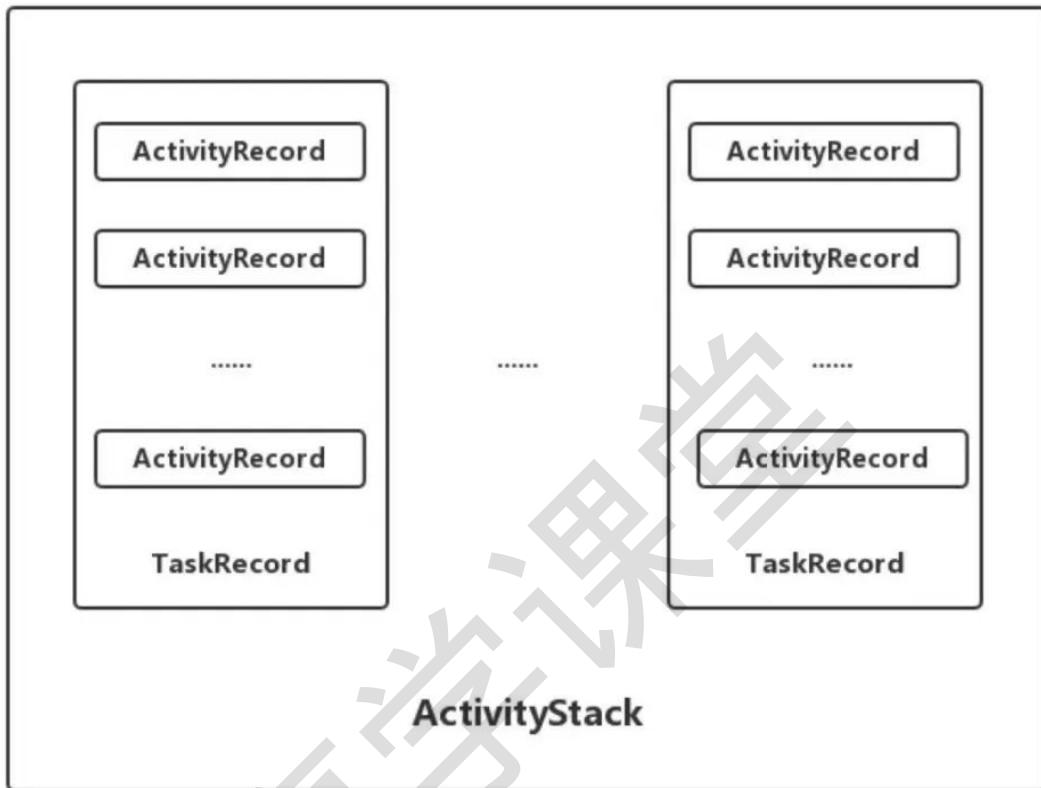
简单介绍一下这行代码：

app是**ProcessRecord**类型，代表一个进程记录，在这就是B进程。
app.thread是**ApplicationThread**类型，上面已经介绍过了，通过它实现进程间通信

上面的app.thread.scheduleLaunchActivity这行代码的意思就是通知B进程中的ApplicationThread执行scheduleLaunchActivity方法，至此工作重心就离开系统AMS进程，剩下的事情全部都是在B进程中执行了。

代码看到这里大脑细胞很可能已经不够用了，因此一下子涌出了好几个类：ActivityStarter、ActivityStackSupervisor、ActivityStack、TaskRecord、ActivityRecord。接下来我们再理一下这几个类之间的关系。

先来张简单的关系图：



ActivityStack、TaskRecord、ActivityRecord关系图.png

算法详解

- 一个**ActivityRecord**对应一个Activity，保存了一个Activity的所有信息；但是一个Activity可能会有多个**ActivityRecord**，因为Activity可以被多次启动，这个主要取决于其启动模式。
- 一个**TaskRecord**由一个或者多个**ActivityRecord**组成，这就是我们常说的任务栈，具有后进先出的特点。
- **ActivityStack**则是用来管理**TaskRecord**的，包含了多个**TaskRecord**。

ActivityRecord

ActivityRecord，源码中的注释介绍：An entry in the history stack, representing an activity.

翻译：历史栈中的一个条目，代表一个activity。

frameworks/base/services/core/java/com/android/server/am/
ActivityRecord.java

```
final class ActivityRecord extends  
ConfigurationContainer implements  
AppwindowContainerListener {  
  
    final ActivityManagerService service; // owner  
    final IApplicationToken.Stub appToken; // window  
    manager token  
    AppwindowContainerController  
    mwindowContainerController;  
    final ActivityInfo info; // all about me  
    final ApplicationInfo appInfo; // information  
    about activity's app  
  
    //省略其他成员变量  
  
    //ActivityRecord所在的TaskRecord  
    private TaskRecord task; // the task this  
    is in.  
  
    //构造方法，需要传递大量信息  
    ActivityRecord(ActivityManagerService _service,  
    ProcessRecord _caller, int _launchedFromPid,  
                int _launchedFromUid, String  
    _launchedFromPackage, Intent _intent, String  
    _resolvedType,  
                ActivityInfo aInfo, Configuration  
    _configuration,  
  
    com.android.server.am.ActivityRecord _resultTo, String  
    _resultWho, int _requestCode,  
                boolean _componentSpecified,  
    boolean _rootVoiceInteraction,  
                ActivityStackSupervisor  
    supervisor, ActivityOptions options,
```

```
com.android.server.am.ActivityRecord sourceRecord) {  
    }  
}
```

- 实际上，`ActivityRecord`中存在着大量的成员变量，包含了一个`Activity`的所有信息。
- `ActivityRecord`中的成员变量`task`表示其所在的`TaskRecord`，由此可以看出：`ActivityRecord`与`TaskRecord`建立了联系。

`startActivity()`时会创建一个`ActivityRecord`：

```
class ActivityStarter {  
    private int startActivity(IApplicationThread  
caller, Intent intent, Intent ephemeralIntent,  
                           String resolvedType,  
ActivityInfo aInfo, ResolveInfo rInfo,  
  
IVoiceInteractionSession voiceSession, IVoiceInteractor  
voiceInteractor,  
                           IBinder resultTo,  
String resultWho, int requestCode, int callingPid, int  
callingUid,  
                           String callingPackage,  
int realcallingPid, int realcallingUid, int startFlags,  
                           ActivityOptions  
options, boolean ignoreTargetSecurity, boolean  
componentSpecified,  
  
com.android.server.am.ActivityRecord[] outActivity,  
TaskRecord inTask) {  
  
    //其他代码略  
  
    ActivityRecord r = new  
ActivityRecord(mService, callerApp, callingPid,  
callinguid,
```

```
        callingPackage, intent, resolvedType,
        aInfo, mService.getGlobalConfiguration(),
        resultRecord, resultWho, requestCode,
        componentSpecified, voiceSession != null,
        mSupervisor, options, sourceRecord);

        //其他代码略
    }
}
```

TaskRecord

`TaskRecord`, 内部维护一个`ArrayList<ActivityRecord>`用来保存`ActivityRecord`。

```
final class TaskRecord extends ConfigurationContainer
implements TaskWindowContainerListener {
    final int taskId;           //任务ID
    final ArrayList<ActivityRecord> mActivities;   //使用一个ArrayList来保存所有的ActivityRecord
    private ActivityStack mStack; //TaskRecord所在的ActivityStack

    //构造方法
    TaskRecord(ActivityManagerService service, int
    _taskId, ActivityInfo info, Intent _intent,
               IVoiceInteractionSession
               _voiceSession, IVoiceInteractor _voiceInteractor, int
               type) {

    }

    //添加Activity到顶部
    void
    addActivityToTop(com.android.server.am.ActivityRecord r)
    {
        addActivityAtIndex(mActivities.size(), r);
    }
```

```
//添加Activity到指定的索引位置
void addActivityAtIndex(int index, ActivityRecord
r) {
    //...

    r.setTask(this); //为ActivityRecord设置
    TaskRecord, 就是这里建立的联系

    //...

    index = Math.min(size, index);
    mActivities.add(index, r); //添加到mActivities

    //...
}

//其他代码略
}
```

- 可以看到TaskRecord中使用了一个ArrayList来保存所有的ActivityRecord。
- 同样，TaskRecord中的mStack表示其所在的ActivityStack。

startActivity()时也会创建一个TaskRecord：

```
class ActivityStarter {

    private int
setTaskFromReuseOrCreateNewTask(TaskRecord
taskToAffiliate, int preferredLaunchStackId,
ActivityStack topStack) {
        mTargetStack =
computeStackFocus(mStartActivity, true, mLaunchBounds,
mLaunchFlags, mOptions);

        if (mReuseTask == null) {
```

```
//创建一个createTaskRecord, 实际上是调用
ActivityStack里面的createTaskRecord () 方法, ActivityStack下
面会讲到
    final TaskRecord task =
mTargetStack.createTaskRecord(
    mSupervisor.getNexttaskIdForUserLocked(mStartActivity.use
    rId),
        mNewTaskInfo != null ?
    mNewTaskInfo : mStartActivity.info,
        mNewTaskIntent != null ?
    mNewTaskIntent : mIntent, mVoiceSession,
        mVoiceInteractor,
!mLaunchTaskBehind /* toTop */,
mStartActivity.mActivityType);

        //其他代码略
    }
}
}
```

ActivityStack

ActivityStack, 内部维护了一个ArrayList<TaskRecord>, 用来管理
TaskRecord。

```
class ActivityStack<T extends StackwindowController>
extends ConfigurationContainer implements
StackwindowListener {

    private final ArrayList<TaskRecord> mTaskHistory
= new ArrayList<>();//使用一个ArrayList来保存TaskRecord

    final int mStackId;

    protected final ActivityStackSupervisor
mStackSupervisor;//持有一个ActivityStackSupervisor, 所有的运
行中的ActivityStacks都通过它来进行管理
```

```
//构造方法

ActivityStack(ActivityStackSupervisor.ActivityDisplay
display, int stackId,
                ActivityStackSupervisor supervisor,
RecentTasks recentTasks, boolean onTop) {

}

TaskRecord createTaskRecord(int taskId,
ActivityInfo info, Intent intent,

IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor,
                boolean toTop, int
type) {

    //创建一个task
    TaskRecord task = new TaskRecord(mService,
taskId, info, intent, voiceSession, voiceInteractor,
type);

    //将task添加到ActivityStack中去
    addTask(task, toTop, "createTaskRecord");

    //其他代码略

    return task;
}

//添加Task
void addTask(final TaskRecord task, final boolean
toTop, String reason) {

    addTask(task, toTop ? MAX_VALUE : 0, true /*
schedulePictureInPictureModeChange */, reason);
}
```

```
//其他代码略  
}  
  
//添加Task到指定位置  
void addTask(final TaskRecord task, int position,  
boolean schedulePictureInPictureModeChange,  
            String reason) {  
    mTaskHistory.remove(task); //若存在，先移除  
  
    //...  
  
    mTaskHistory.add(position, task); //添加task到  
mTaskHistory  
    task.setStack(this); //为TaskRecord设置  
ActivityStack  
  
    //...  
}  
  
//其他代码略  
}
```

- 可以看到ActivityStack使用了一个ArrayList来保存TaskRecord。
- 另外，ActivityStack中还持有ActivityStackSupervisor对象，这个是用来管理ActivityStacks的。

ActivityStack是由ActivityStackSupervisor来创建的，实际ActivityStackSupervisor就是用来管理ActivityStack的，继续看下面的ActivityStackSupervisor分析。

ActivityStackSupervisor

ActivityStackSupervisor，顾名思义，就是用来管理ActivityStack的。

```
public class ActivityStackSupervisor extends ConfigurationContainer implements DisplayListener {  
  
    ActivityStack mHomeStack; //管理的是Launcher相关的任务  
  
    ActivityStack mFocusedStack; //管理非Launcher相关的任务  
  
    //创建ActivityStack  
    ActivityStack createStack(int stackId,  
        ActivityStackSupervisor.ActivityDisplay display, boolean  
        onTop) {  
        switch (stackId) {  
            case PINNED_STACK_ID:  
                //PinnedActivityStack是ActivityStack的子类  
                return new PinnedActivityStack(display, stackId, this, mRecentTasks,  
                    onTop);  
            default:  
                //创建一个ActivityStack  
                return new ActivityStack(display,  
                    stackId, this, mRecentTasks, onTop);  
        }  
    }  
}
```

- `ActivityStackSupervisor` 内部有两个不同的 `ActivityStack` 对象：`mHomeStack`、`mFocusedStack`，用来管理不同的任务。
- `ActivityStackSupervisor` 内部包含了创建 `ActivityStack` 对象的方法。

AMS 初始化时会创建一个 `ActivityStackSupervisor` 对象。

```

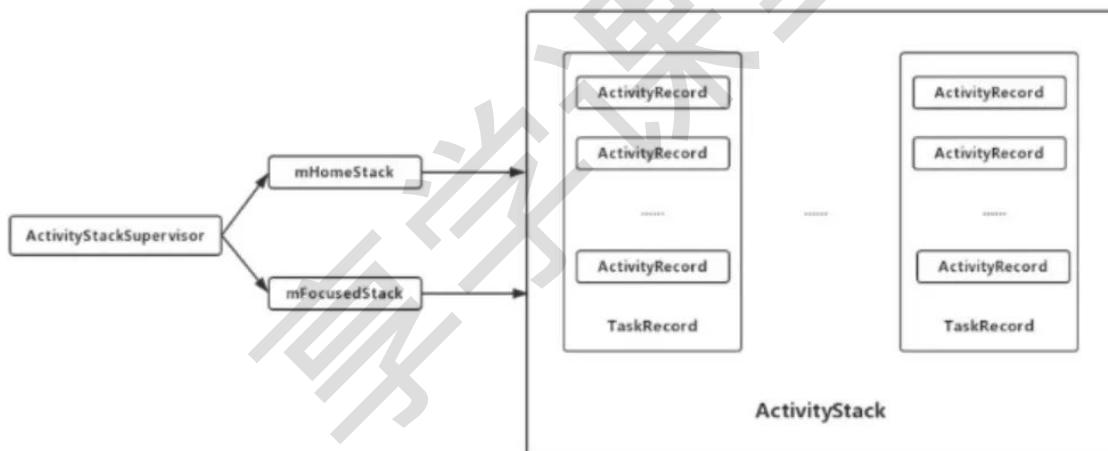
public ActivityManagerService(){
    mStackSupervisor = createStackSupervisor();
}

protected ActivityStackSupervisor createStackSupervisor()
{
    return new ActivityStackSupervisor(this,
mHandler.getLooper());
}

```

总结

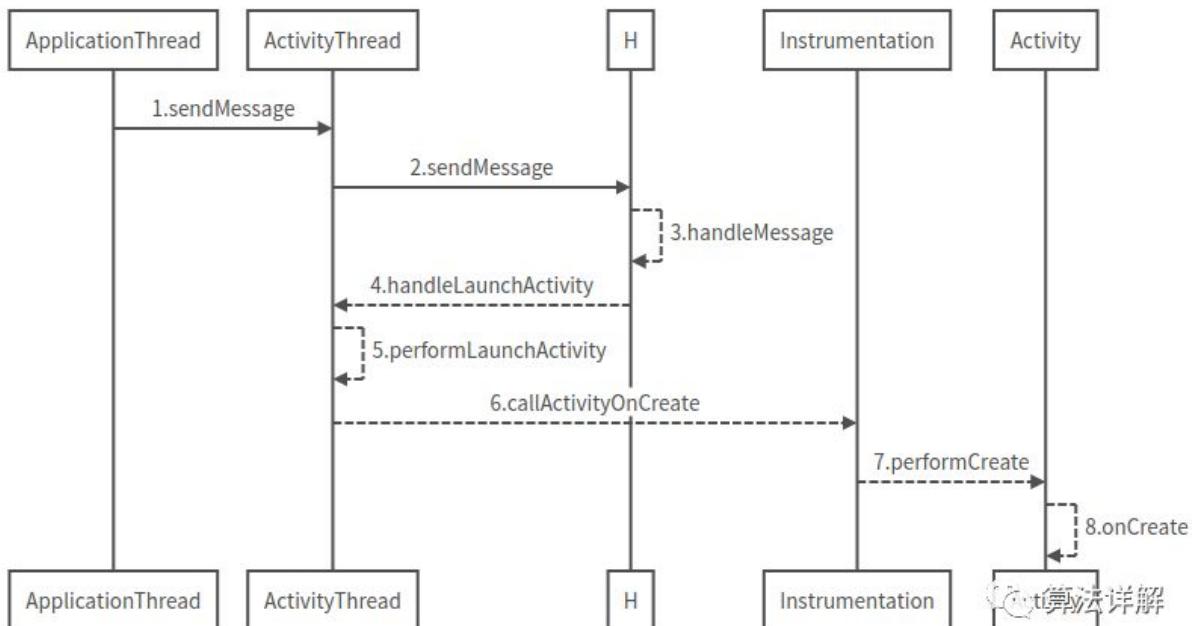
所以，实际上，他们的关系应该是这样的：



ActivityStack、TaskRecord、ActivityRecord、ActivityStackSupervisor关系图.png 算法详解

(三) ApplicationThread --> Activity

同样还是先来一张时序图：



从ApplicationThread到真正启动一个Activity，其实也并不是很复杂。

ApplicationThread的scheduleLaunchActivity

ApplicationThread是ActivityThread的内部类。

scheduleLaunchActivity中就是将启动Activity的参数封装成**ActivityClientRecord**，然后调用**ActivityThread**的sendMessage方法。

```

@Override
    public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
                                              ActivityInfo info, Configuration curConfig, Configuration overrideConfig,
                                              CompatibilityInfo compatInfo, String referrer, IVoiceInteractor voiceInteractor,
                                              int procState, Bundle state, PersistableBundle persistentState,
                                              List<ResultInfo> pendingResults, List<ReferrerIntent> pendingNewIntents,
                                              boolean notResumed, boolean isForward, ProfilerInfo profilerInfo) {
        updateProcessState(procState, false);

```

```
        ActivityClientRecord r = new
ActivityClientRecord();

        r.token = token;
        r.ident = ident;
        r.intent = intent;
        r.referrer = referrer;
        r.voiceInteractor = voiceInteractor;
        r.activityInfo = info;
        r.compatInfo = compatInfo;
        r.state = state;
        r.persistentState = persistentState;

        r.pendingResults = pendingResults;
        r.pendingIntents = pendingNewIntents;

        r.startsNotResumed = notResumed;
        r.isForward = isForward;

        r.profilerInfo = profilerInfo;

        r.overrideConfig = overrideConfig;
        updatePendingConfiguration(curConfig);

        //调用ActivityThread的sendMessage方法，发送
LAUNCH_ACTIVITY信息
        sendMessage(H.LAUNCH_ACTIVITY, r);
    }
}
```

ApplicationThread的sendMessage

```
private void sendMessage(int what, Object obj) {
    sendMessage(what, obj, 0, 0, false);
}

private void sendMessage(int what, Object obj, int
arg1, int arg2, boolean async) {
    if (DEBUG_MESSAGES) Slog.v(
```

```
        TAG, "SCHEDULE " + what + " " +
mH.codeToString(what)
        + ":" + arg1 + " / " + obj);
Message msg = Message.obtain();
msg.what = what;
msg.obj = obj;
msg.arg1 = arg1;
msg.arg2 = arg2;
if (async) {
    msg.setAsynchronous(true);
}
mH.sendMessage(msg);
}
```

H是ActivityThread的一个内部类

```
private class H extends Handler { //继承Handler
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>>
handling: " + codeToString(msg.what));
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {//启动ACTIVITY
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
"activityStart");
                final ActivityClientRecord r =
(ActivityClientRecord) msg.obj;

                r.packageInfo =
getPackageInfoNoCheck(
                    r.activityInfo.applicationInfo, r.compatInfo);
                handleLaunchActivity(r, null,
"LAUNCH_ACTIVITY");
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            } break;
            case RELAUNCH_ACTIVITY: {
```

```
//...
} break;
case PAUSE_ACTIVITY: {
    //...
} break;
case PAUSE_ACTIVITY_FINISHING: {
    //...
} break;
```

可以看到H里面会对各种状态进行处理。LAUNCH_ACTIVITY中会调用handleLaunchActivity，继续往下看。

```
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent, String reason) {
    //...
    Activity a = performLaunchActivity(r, customIntent);

    if (a != null) {
        r.createdConfig = new Configuration(mConfiguration);
        reportSizeConfigurations(r);
        Bundle oldState = r.state;
        //这里最终会调用Activity的onResume生命周期
        handleResumeActivity(r.token, false, r.isForward,
            !r.activity.mFinished &&
            !r.startsNotResumed, r.lastProcessedSeq, reason);
        //...
    }

    private Activity
    performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        //从ActivityClientRecord中获取要启动的Activity信息
        ActivityInfo aInfo = r.activityInfo;
```

```
    if (r.packageInfo == null) {
        r.packageInfo =
getPackageManager(aInfo.applicationInfo, r.compatInfo,
                Context.CONTEXT_INCLUDE_CODE);
    }

    ComponentName component =
r.intent.getComponent();
    if (component == null) {
        component = r.intent.resolveActivity(
            mInitialApplication.getPackageManager());
        r.intent.setComponent(component);
    }

    if (r.activityInfo.targetActivity != null) {
        component = new
ComponentName(r.activityInfo.packageName,
            r.activityInfo.targetActivity);
    }

    //ContextImpl在这里创建, ContextImpl是Context的具体实现
    ContextImpl appContext =
createBaseContextForActivity(r);
    Activity activity = null;

    try {
        java.lang.ClassLoader cl =
appContext.getClassLoader();

        //通过Instrumentation的newActivity使用类加载器来
        //创建Activity对象
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(),
r.intent);

        StrictMode.incrementExpectedActivityCount(activity.getClas
ss());
    }
}
```

```
r.intent.setExtrasClassLoader(c1);
r.intent.prepareToEnterProcess();
if (r.state != null) {
    r.state.setClassLoader(c1);
}
}

try {
    //创建Application对象，如果已经创建过，则不会重复创建，保证一个应用中只有一个Application对象；
    //实际上跟Activity一样也是在Instrumentation中使用类加载器来创建的
    Application app =
r.packageInfo.makeApplication(false, mInstrumentation);

    if (localLOGV) Slog.v(TAG, "Performing launch
of " + r);
    if (localLOGV) Slog.v(
        TAG, r + ": app=" + app
        + ", appName=" + app.getPackageName()
        + ", pkg=" +
r.packageInfo.getPackageName()
        + ", comp=" +
r.intent.getComponent().toShortString()
        + ", dir=" +
r.packageInfo.getAppDir());

    if (activity != null) {
        //...
        //attach中会建立Activity与ContextImpl的联系
        //attach中还会创建window并与window建立联系 可以看Activivity中的源码
        activity.attach(appContext, this,
getInstrumentation(), r.token,
            r.ident, app, r.intent,
r.activityInfo, title, r.parent,
```

```
        r.embeddedID,
r.lastNonConfigurationInstances, config,
        rreferrer, r.voiceInteractor,
window, r.configCallback);

        //...

        if (r.isPersistable()) {
            //回调Activity的OnCreate

mInstrumentation.callActivityOnCreate(activity, r.state,
r.persistentState);
        } else {
            //回调Activity的OnCreate

mInstrumentation.callActivityOnCreate(activity, r.state);
        }

        //...
        return activity;
}
```

可以看出，最终调用了Instrumentation的callActivityOnCreate方法，感觉离目的地越来越近了！

Instrumentation的callActivityOnCreate

```
public void callActivityOnCreate(Activity activity,
Bundle icicle,
        PersistableBundle persistentState) {
    prePerformCreate(activity);
    activity.performCreate(icicle, persistentState);
    postPerformCreate(activity);
}
```

Activity的performCreate

```
final void performCreate(Bundle icicle) {
    performCreate(icicle, null);
```

```

    }

    final void performCreate(Bundle icicle,
    PersistableBundle persistentState) {
        mCanEnterPictureInPicture = true;
        restoreHasCurrentPermissionRequest(icicle);
        if (persistentState != null) {
            onCreate(icicle, persistentState); //onCreate
        } else {
            onCreate(icicle); //onCreate
        }
        mActivityTransitionState.readState(icicle);

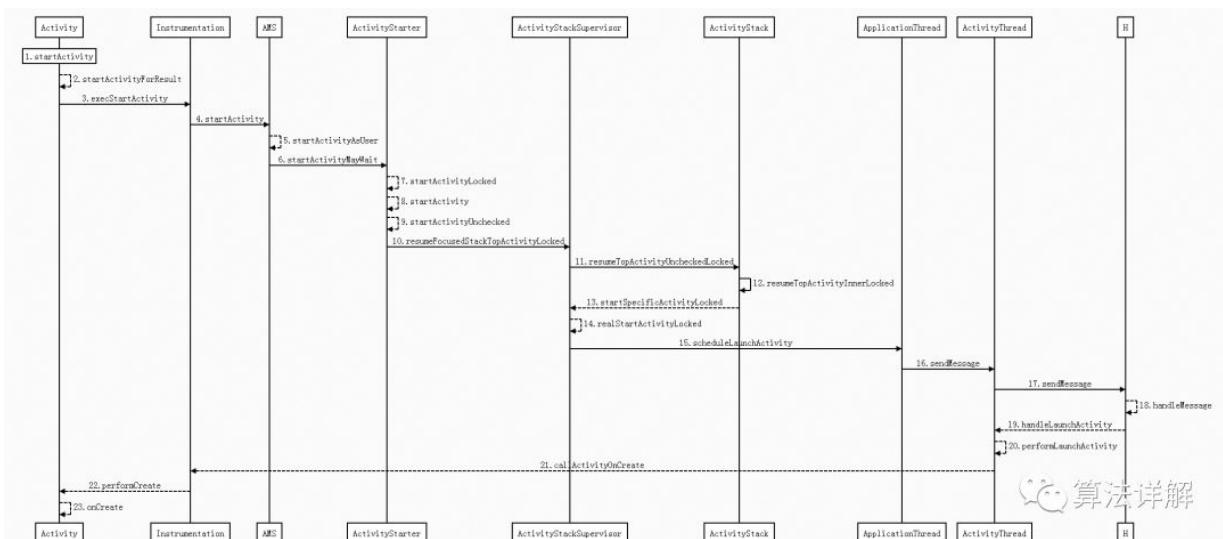
        mVisibleFromClient =
        !mWindow.getWindowStyle().getBoolean(
            com.android.internal.R.styleable.Window_windowNoDisplay,
            false);
        mFragments.dispatchActivityCreated();

        mActivityTransitionState.setEnterActivityOptions(this,
            getActivityOptions());
    }
}

```

可以看到，onCreate被调用起来了。至此，Activity被启动起来了。

最后，放张总的时序图（图太大，请另外打开或保存）：

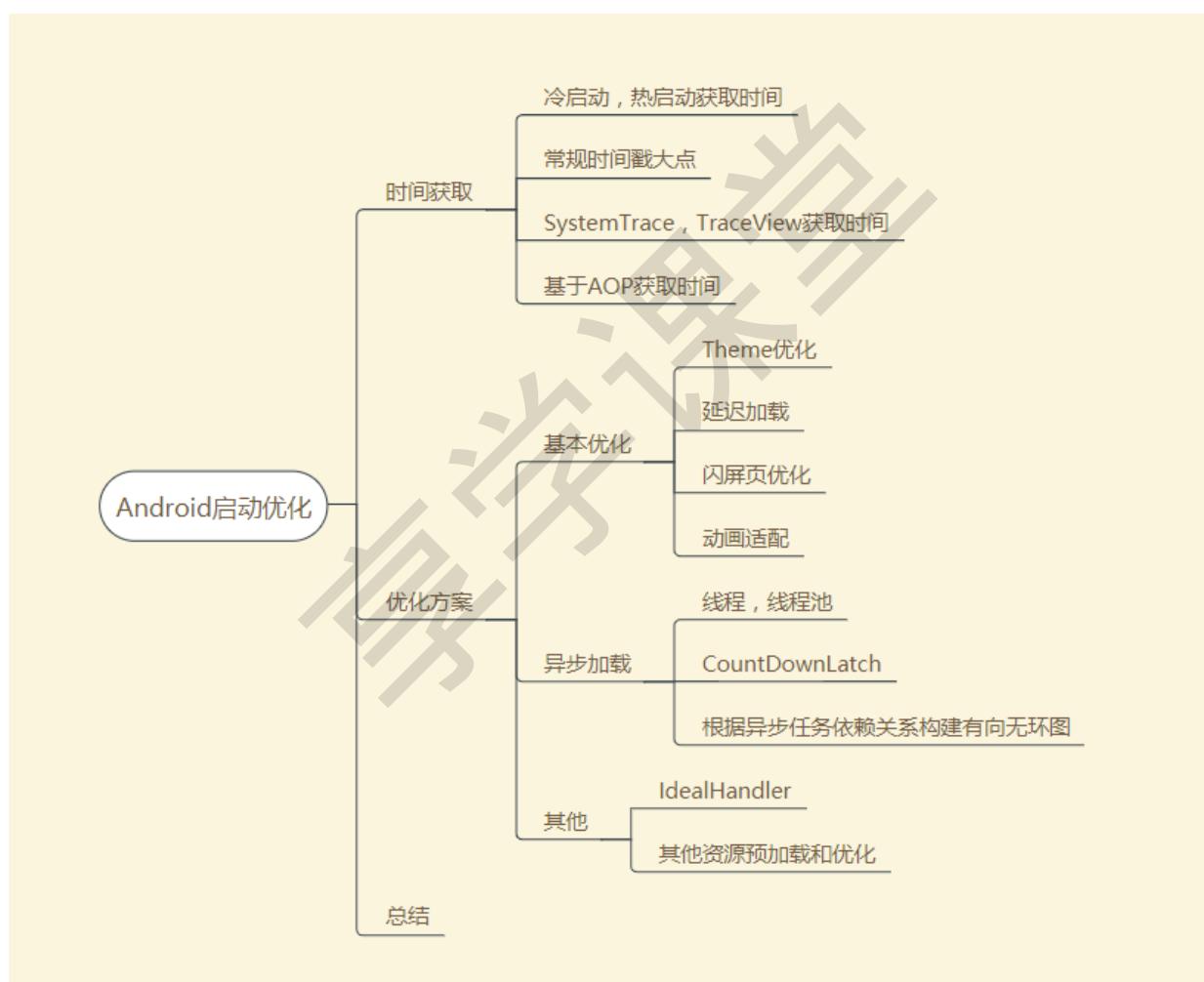


1.2 Android性能优化--启动优化

1.2.1前言

一个应用App的启动速度能够影响用户的首次体验，启动速度较慢(感官上)的应用可能导致用户再次开启App的意图下降，或者卸载放弃该应用程序。本文会通过以下几个方面来介绍应用启动的相关指标和优化，提供应用的启动速度。

整体文章思路如下：



1.2.2冷启动&热启动

通常来说，启动方式分为两种：冷启动和热启动。

1. 冷启动：当启动应用时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用，这个启动方式就是冷启动。

2. 热启动：当启动应用时，后台已有该应用的进程（例：按back键、home键，应用虽然会退出，但是该应用的进程是依然会保留在后台，可进入任务列表查看），所以在已有进程的情况下，这种启动会从已有的进程中来启动应用，这种方式叫热启动。

两者之间的特点如下：

- 冷启动：系统会重新创建一个新的进程分配给该应用，从Application创建到UI绘制等相关流程都会执行一次。
- 热启动：应用还在后台，因此该启动方式不会重建Application，只会重新绘制UI等相关流程。

冷热启动时间的计算命令：

```
adb shell am start -W  
[packageName]/[packageName.XxxActivity]
```

- 参数说明：
 - 1、ThisTime：一般和TotalTime时间一样。除非在应用启动时开了一个透明的Activity预先处理一些事再显示出主Activity，这样将比TotalTime小。
 - 2、TotalTime：应用的启动时间。包含创建进程+Application初始化+Activity初始化到界面显示。
 - 3、WaitTime：一般比TotalTime大点，包含系统影响的耗时。

对于我们的应用来说结果如下：

- 冷启动

```
C:\Users\xiuwen.zhan>adb shell am start -W com.android.browser/.BrowserActivity  
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.android.browser/.BrowserActivity }  
Status: ok  
Activity: com.android.browser/.BrowserActivity  
ThisTime: 2040  
TotalTime: 2040  
WaitTime: 2093  
Complete
```

冷启动.png

- 热启动

```
C:\Users\xiuwen.zhan>adb shell am start -W com.android.browser/.BrowserActivity  
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.android.browser/.BrowserActivity }  
Warning: Activity not started, its current task has been brought to the front  
Status: ok  
Activity: com.android.browser/.BrowserActivity  
ThisTime: 357  
TotalTime: 357  
WaitTime: 391  
Complete
```

热启动.png

可以看到两者时间相差比较大。

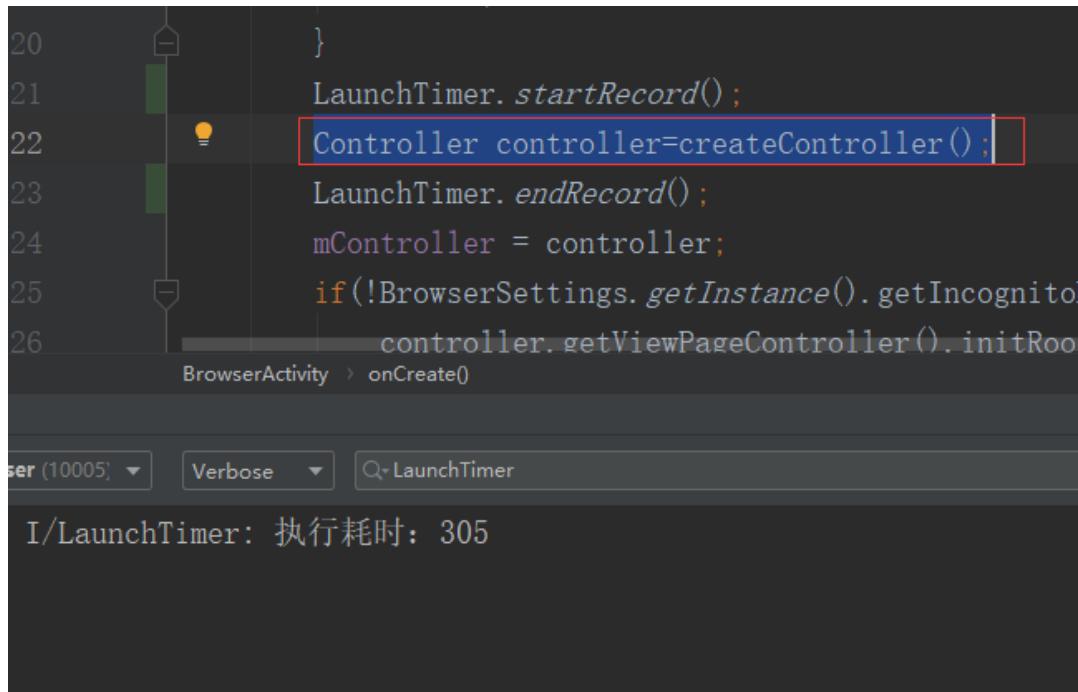
根据该命令基本可以看出一个应用的启动速度了，从冷启动热启动的相关关系，当我们需要优化启动速度的时候，优化冷启动速度即可。但是该命令我们只是大概知道应用的启动速度，但并不知道我们的应用具体哪个位置耗时，影响启动速度，后续我会介绍如何获取启动具体耗时时间。

1.2.3 常规获取时间方法

常规获取时间方法无非就是在方法执行前记录下时间，在方法执行完毕后记录时间，两者时间之差就是该方法执行的时间，封装一个基础类如下：

```
public class LaunchTimer {  
  
    private static final String TAG = "LaunchTimer";  
    private static long sTime;  
  
    public static void startRecord() {  
        sTime = System.currentTimeMillis();  
    }  
  
    public static void endRecord() {  
        long cost = System.currentTimeMillis() - sTime;  
        NLog.i(TAG, "执行耗时: %s", cost);  
    }  
}
```

使用方式如下，可以直观的看出createController方法执行的时间



createController耗时.png

这样已经很直观了，可以具体到该方法的执行时间，如果要继续分析则对该方法内部继续执行该代码即可。但是这里有一个问题如果要知道10个或者更多方法的执行时间，这个方法看起来是可以，但写起来过于繁琐，且不符合程序员的习惯，关于这种场景后面会介绍如何处理。

1.2.4 TraceView和SysTrace工具使用

- TraceView使用：TraceView是Android平台配备一个很好的性能分析工具，它可以通过图形化的方式让我们了解我们要跟踪的程序的性能，并且能具体到方法。

使用方式：

1. 通过Android studio自带的traceview查看 (Android profiler) 。
2. 通过Android SDK自带的Debug。
3. 通过DDMS中的traceview查看。

本文主要介绍第二种方式，通过sdk中的方法，对应用进行打点获取相关信息：

```
@Override  
public void onCreate(Bundle icicle) {  
    setTheme(R.style.BrowserTheme);  
    Intent intent = getIntent();  
    NLog.i(LOGTAG, "onCreate");
```

```

super.onCreate(icicle);
//开始记录，且该方法可以设置文件大小和路径
Debug.startMethodTracing("browser.trace");
Controller controller=createController();
mController = controller;

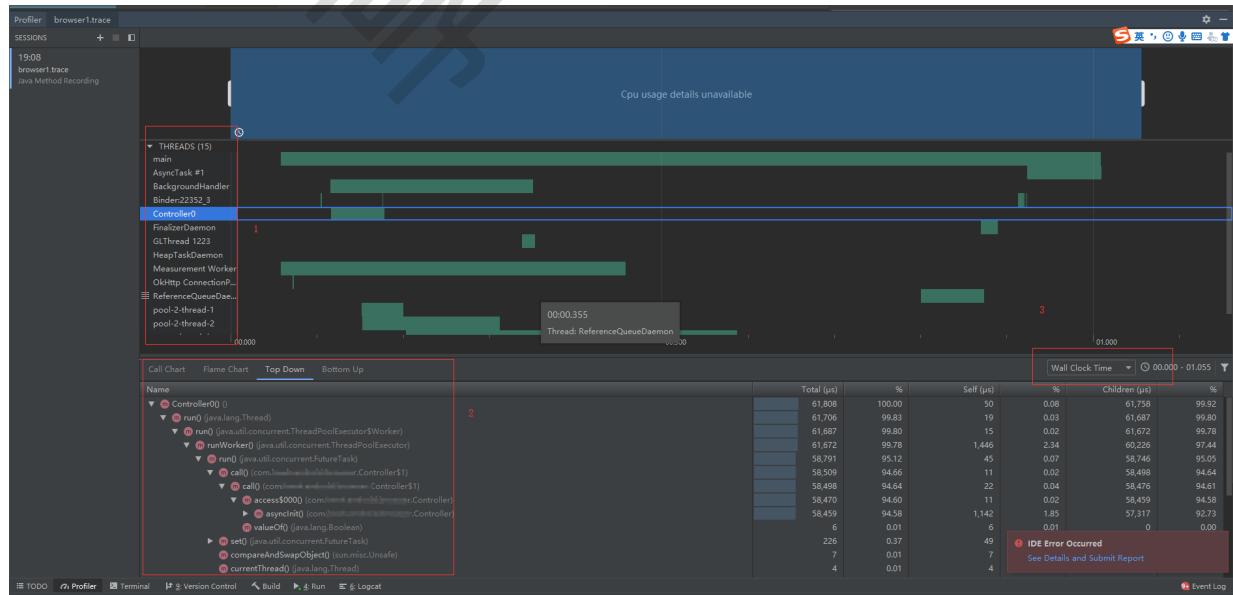
getWindow().getDecorView().setSystemUiVisibility(
    view.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN|view.SYSTEM_UI_FLAG
    _LIGHT_STATUS_BAR);
    controller.handleThirdPartyIntent(intent);
//结束记录
Debug.stopMethodTracing();
}

```

如上可以在目录下可以生成如下文件

```
/sdcard/Android/data/com.xxx.xx.browser/files/browser.trace
```

导出改文件，通过Android Studio的profile打开改文件



traceview.png

1处可以看出有多少线程。

2处可以看出具体方法的耗时。

3处有两个选项：

- wall clock time:代码在线程上执行的真正时间[有一部分是等待cpu轮询时间]
- thread time :cpu执行的时间
一般是优化的是cpu执行时间

关于该图如何查看和阅读可以参照文章[Android性能优化—TraceView的使用](#)

结合业务代码走查发现Controller0线程为一个线程，因此主线程一些操作可以放进去执行，从而减少main线程的耗时。走查代码可以发现在Controller0线程中执行如下

```
requestPermission();  
ExecutorService service =  
Executors.newSingleThreadExecutor(new  
NamedThreadFactory("Controller"));  
Future<Boolean> future = service.submit(new  
Callable<Boolean>() {  
    @Override  
    public Boolean call() throws Exception {  
        try {  
            asyncInit();  
  
        } catch (Exception e) {  
            return false;  
        }  
        return true;  
    }  
});
```

requestPermission()方法执行在main线程中，因此我们可以把其放在Controller0线程中执行，从而减少main线程的的时间

```
ExecutorService service =
Executors.newSingleThreadExecutor(new
NamedThreadFactory("Controller"));
    Future<Boolean> future = service.submit(new
Callable<Boolean>() {
    @Override
    public Boolean call() throws Exception {
        try {
            requestPermission();
            asyncInit();
        } catch (Exception e) {
            return false;
        }
        return true;
    }
});
```

经测试发现无问题，且对比此时的trace文件发现修改前后main线程时间相对来说减少很多。

- SysTrace使用：

SysTrace用于收集可帮助您检查原生系统进程的详细系统级数据，例如CPU调度、磁盘活动、应用线程等，并解决掉帧引起的界面卡顿。

使用方式：

在代码的开始位置加上tag

```
TraceCompat.beginSection("AppOnCreate");
```

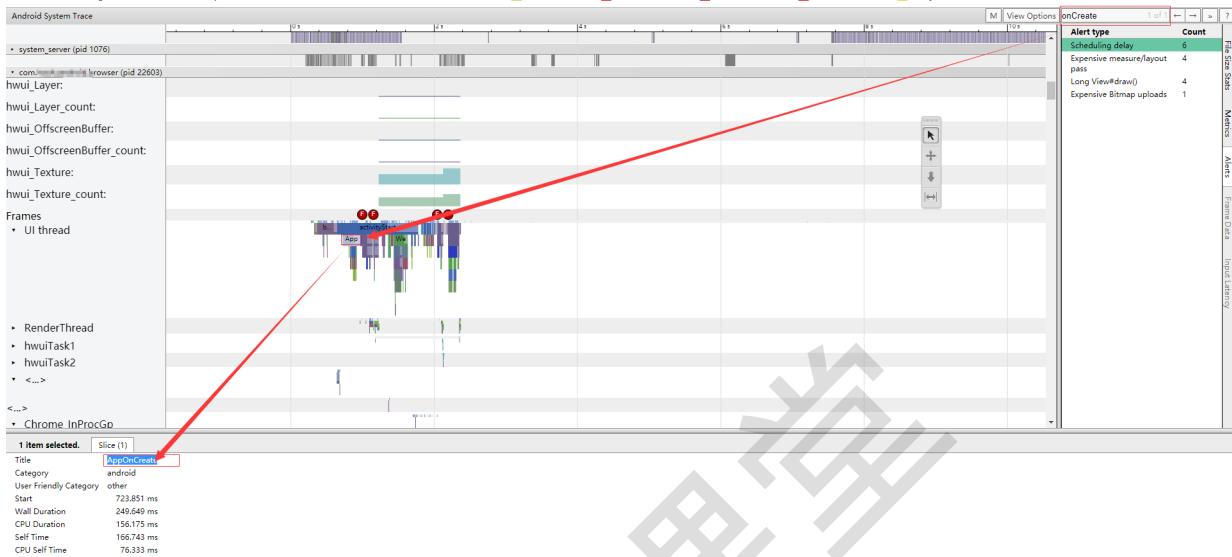
然后指定位置结束

```
TraceCompat.endSection();
```

，即可以抓取到整个应用在此过程的相关信息，例如在onCreate方法中添加上述两行代码，执行相关python命令：

```
python systrace.py -b 32768 -t 10 -a com.xxx.xxx.browser  
-o browser.html sched gfx view wm am app
```

操作相关应用，即可以抓取整个过程的相关信息：



systrace.png

即可以看到添加的tag“AppOnCreate”，对应的时间信息：

1. Wall Duration 代表的方法从开始到结束的耗时
2. CPU Duration 代表CPU的执行时间

通过这两个参数可以看出此流程中执行的时间等相关信息

我们都知道CPU是轮询模式，因此优化的方向可以说是两个方向，提高CPU的核数和优化CPU执行的时间。

关于Systrace的具体使用及命令可以参照

[性能优化工具（二）-Systrace](#)

[了解 Systrace](#)

1.2.5 通过AOP获取时间

- AOP:面向切面编程(Aspect-Oriented Programming)。如果说，OOP如果是把问题划分到单个模块的话，那么AOP就是把涉及到众多模块的某一类问题进行统一管理。打个比方Android 里面PMS，AMS都拥

有各自的职责，但是他们都需要通过log系统管理log，这就是一种AOP思想。

AspectJ实际上是对AOP编程思想的一个实践，当然，除了AspectJ以外，还有很多其它的AOP实现，例如ASMDex，但目前最好、最方便的，依然是AspectJ。

AspectJ的使用如下：

根目录gradle下引用：

```
classpath 'com.hujiang.aspectjx:gradle-android-plugin-aspectjx:2.0.0'
```

app目录gradle文件下引用：

```
implementation 'org.aspectj:aspectjrt:1.8.+'
```

此两处引用完成之后，就是代码编写：

```
package com.xx.xxx.browser.aspect;

import android.util.Log;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

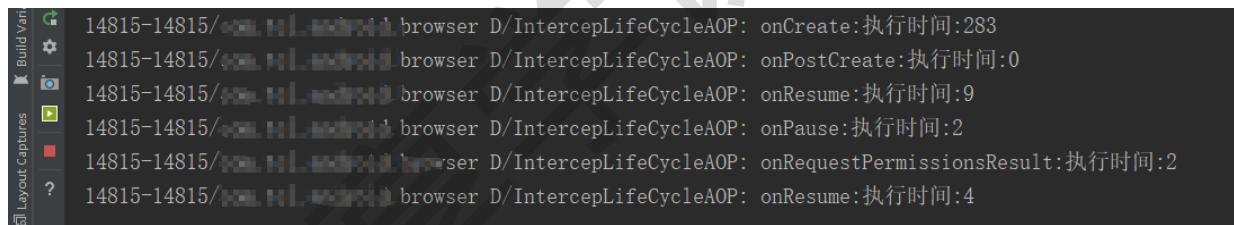
@Aspect
public class IntercepLifeCycleAOP {
    //获取该Activity下的所有on开头的方法耗时
    @Around("execution(* com.xxx.xxx.BrowserActivity.on**(..))")
    public Object getTime(ProceedingJoinPoint joinPoint)
    {
        Object proceed = null;
```

```

        long start = System.currentTimeMillis();
        try {
            proceed = joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        long end = System.currentTimeMillis();
        Log.d("IntercepLifecycleAOP",
joinPoint.getSignature().getName() + ":执行时间:" + (end -
start));
        return proceed;
    }
}

```

引入之后结果如下：



AspectJ.png

可以看到具体方法的耗时。

采用注解方式，其中 `Around` 需要有一定的 AspectJ 相关的语法，具体参考：

[AspectJ 在 Android 中的使用](#)

[gradle plugin android aspectjx](#)

[hugo](#)

1.2.6 用户体验优化

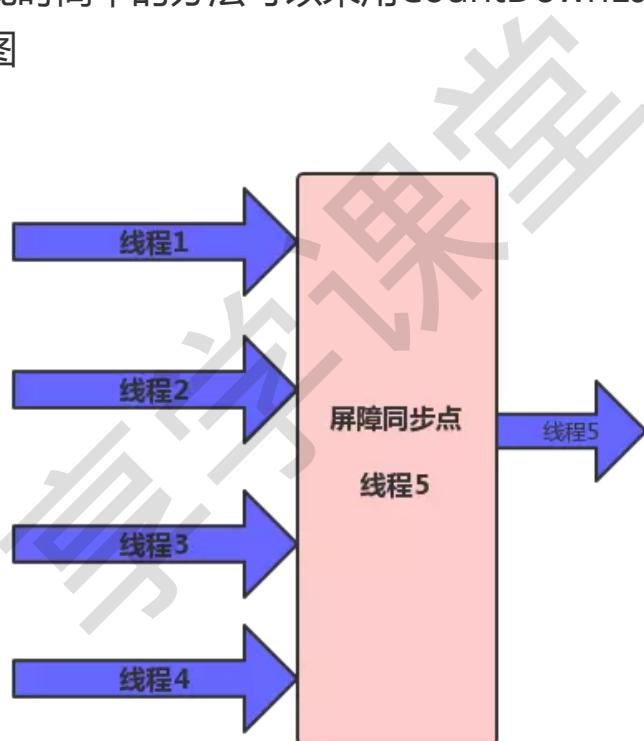
1. 主题优化：通过给应用设置一个透明主题，在应用启动完成之后，再给其赋予本该有的主题，通过对启动页的主题设置后，就会将白屏/黑

屏抹去，用户点击App的图标就展示启动图，让用户先产生启动很快的“错觉”。

2. 动画兼容：根据不同年代的机型可以选择执行或不执行相关动画，或者延迟其他相关操作，可根据[device-year-class](#)来判断具体年份
3. UI布局优化。

1.2.7 异步加载

1. 采用线程加载一些资源，比如sdk初始化，配置信息拉取等相关资源。线程，线程池，IntentServices均可以，配合延迟效果更好。
2. 当我们采用线程之间的可能会存在各线程之间相互等待依赖等相关问题，资源A线程必须在资源B加载完成，才能加载，但两者又会在不同的线程之间，此时简单的办法可以采用CountDownLatch来实现。其整体思路如下图



CountDownLatch.png

具体使用方法可以参照文章[Android 并发之CountDownLatch、CyclicBarrier的简单应用（三）](#)

3. 使用 Pipeline 机制，根据业务优先级规定业务初始化时机，制定启动框架，它们为各个任务建立依赖关系，最终构成一个有向无环图。对于可以并发的任务，会通过线程池最大程度提升启动速度。无论是微信的[mmkernel](#) 还是阿里的[Alpha](#) 都具备这种能力。

4. 其他方案：

除了上述几种，我们也可以利用IdealHandler，dex分包等相关方式做到启动优化。

1.2.8 总结

上面主要介绍了如何获取启动的相关事件和相关优化知识点。关于时间就是尽量使用工具，关于优化整体思路就是能预加载能延迟加载的资源尽量去预加载去延迟加载，能异步的业务尽量异步。

当然优化这个话题也是要根据具体的业务逻辑来定，总之：

对于启动优化要警惕 KPI 化，我们要解决的不是一个数字，而是用户真正的体验问题。

上述只是提供一些思路和方式，还有很多奇淫技巧，欢迎给位大佬评论指出。

第二节 内存优化

2.1 Android性能优化：这是一份全面&详细的内存优化指南

前言

- 在 Android 开发中，性能优化策略十分重要
- 本文主要讲解性能优化中的**内存优化**，希望你们会喜欢

2.1.1 定义

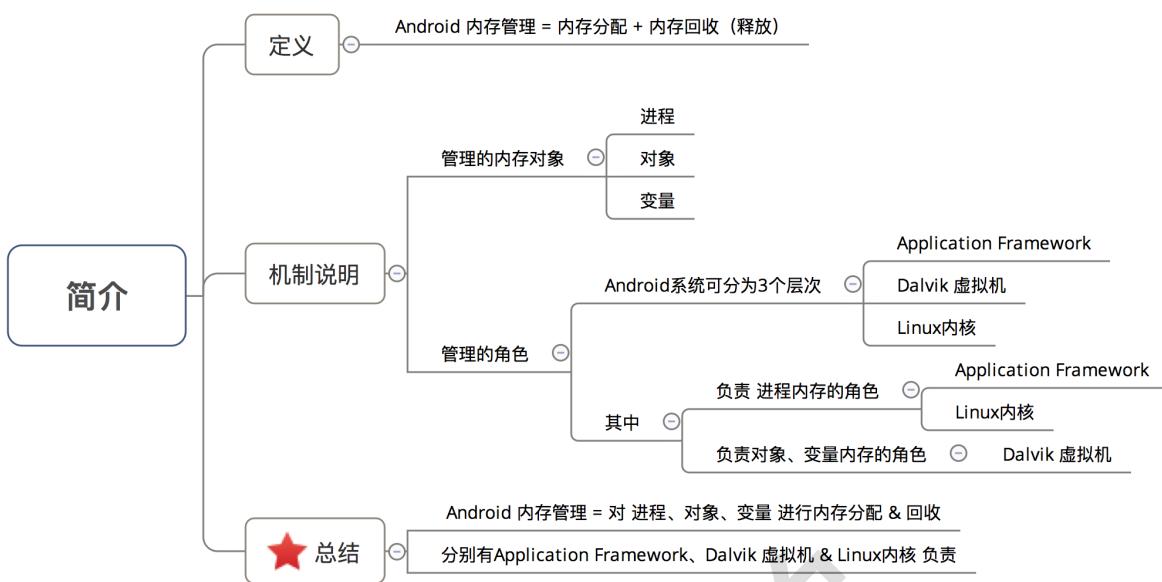
优化处理 应用程序的**内存使用、空间占用**

2.1.2 作用

避免因不正确使用内存 & 缺乏管理，从而出现 **内存泄露 (ML)**、**内存溢出 (OOM)**、**内存空间占用过大** 等问题，最终导致应用程序崩溃 (Crash)

2.1.3 储备知识：Android 内存管理机制

3.1 简介



下面，将针对回收 进程、对象、变量 的内存分配 & 回收进行详细讲解

3.2 针对进程的内存策略

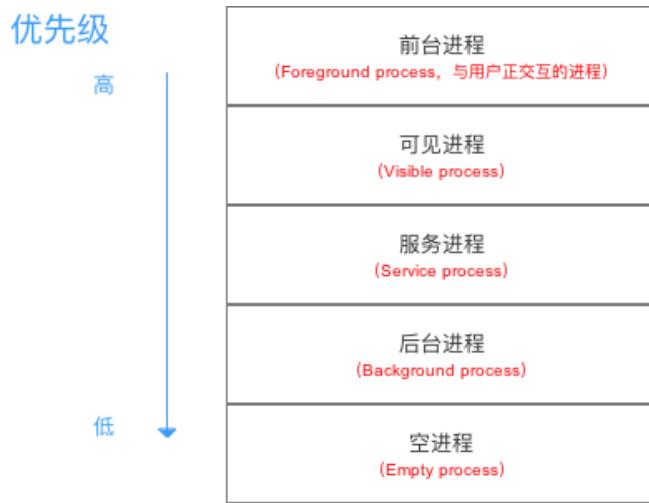
a. 内存分配策略

由 `ActivityManagerService` 集中管理 所有进程的内存分配

b. 内存回收策略

- 步骤1：`Application Framework` 决定回收的进程类型
Android 中的进程 是托管的；当进程空间紧张时，会 按进程优先级低->高的顺序 自动回收进程

| Android 将进程分为 5 个优先等级，具体如下：



注：当系统需回收进程时，最先回收空进程、最后回收前台进程

- 步骤2：`Linux` 内核真正回收具体进程

1. `ActivityManagerService` 对所有进程进行评分（评分存放在变量 `adj` 中）
2. 更新评分到 `Linux` 内核
3. 由 `Linux` 内核完成真正的内存回收

此处仅总结流程，这其中的过程复杂，有兴趣的读者可研究系统源码

`ActivityManagerService.java`

3.3 针对对象、变量的内存策略

- `Android` 的对于对象、变量的内存策略同 `Java`
- 内存管理 = 对象 / 变量的内存分配 + 内存释放

下面，将详细讲解**内存分配 & 内存释放策略**

a. 内存分配策略

- 对象 / 变量的内存分配 **由程序自动负责**
- 共有3种：静态分配、栈式分配、& 堆式分配，分别面向静态变量、局部变量 & 对象实例
- 具体介绍如下

| 内存分配策略 | 使用的内存空间 | 存储的数据 | 分配策略 描述 |
|------------------|----------------|--|---|
| 静态分配 | 方法区 (静态存储区) | 存储 已被虚拟机加载的 类信息、常量、静态变量 | 在程序编译时就已分配好 & 存在于程序整个运行期间 (不需回收) |
| 栈式分配 | 栈区 (Stack) | 存储方法执行时的局部变量 (含数据类型、对象的引用) (以帧栈的形式) | <ul style="list-style-type: none"> 方法执行时, 定义局部变量 则由程序自动在栈中分配内存 方法执行结束 / 超出变量域时, 则由帧栈自动释放该部分内存 效率高 (因 栈内存分配运算置于处理器的指令集中) 但分配的内存容量有限 |
| 堆式分配 (动态内存分配) | 堆区 (Heap) | 存储Java对象的实例 & 实例内的成员变量 <ul style="list-style-type: none"> 即 采用关键字new 出来的对象 实例的成员变量 = 基本数据类型、引用 & 引用的对象实体 | <ul style="list-style-type: none"> 创建对象实例时, 由程序分配 (由Java垃圾回收管理器 自动管理: 不使用时则回收) 访问方式 <ol style="list-style-type: none"> 在堆中创建1个对象 / 数组 & 在栈中定义一个特殊的变量 (引用变量) = 数组 / 对象在堆内存中的首地址 通过 引用变量 来访问堆内存中的对象 / 数组 |

注：用1个实例讲解 内存分配

```

public class Sample {
    // 该类的实例对象的成员变量s1、mSample1 & 指向对象存放在堆内存中
    int s1 = 0;
    Sample mSample1 = new Sample();

    // 方法中的局部变量s2、mSample2存放在 栈内存
    // 变量mSample2所指向的对象实例存放在 堆内存
    public void method() {
        int s2 = 0;
        Sample mSample2 = new Sample();
    }
}

// 变量mSample3的引用存放在栈内存中
// 变量mSample3所指向的对象实例存放在堆内存
// 该实例的成员变量s1、mSample1也存放在堆内存中
Sample mSample3 = new Sample();

```

b. 内存释放策略

- 对象 / 变量的内存释放 由 Java 垃圾回收器 (GC) / 帧栈 负责
- 此处主要讲解对象分配 (即堆式分配) 的内存释放策略 = Java 垃圾回收器 (GC)

由于静态分配不需释放、栈式分配仅 通过帧栈自动出、入栈，较简单，故不详细描述

- Java 垃圾回收器 (GC) 的内存释放 = 垃圾回收算法，主要包括：



- 具体介绍如下

| 算法名称 | 算法思想 | 优点 | 缺点 | 应用场景 |
|------------|---|---|--|---|
| 标记-清除 算法 | <ul style="list-style-type: none"> 标记阶段：标记出所有需要回收的对象； 清除阶段：统一清除（回收）所有被标记的对象。 | 实现简单 | <ul style="list-style-type: none"> 效率问题：标记和清除两个过程效率不高 空间问题：标记 - 清除后，会产生大量不连续的内存碎片。 | 对象存活率较低 & 垃圾回收行为频率低 <small>(如老年代)</small> |
| 复制 算法 | <ul style="list-style-type: none"> 将内存分为大小相等的两块，每次使用其中一块； 当使用的这块内存用完，就将这块内存上还存活的对象复制到另一块还没有用过的内存上； 最终将使用的那块内存一次清理掉。 | <ul style="list-style-type: none"> 解决了标记-清除算法中 清除效率低的问题：每次仅回收内存的一半区域。 解决了标记-清除算法中 空间产生不连续内存碎片的问题：将已使用内存上的存活对象 移动到栈顶的指针，按顺序分配内存即可。 | <ul style="list-style-type: none"> 每次使用的内存缩小为原来的一半。 当对象存活率较高的情况下要做很多复制操作，即效率会变低 | 对象存活率较低 & 需要频繁进行垃圾回收 的区域 <small>(如新生代)</small> |
| 标记 - 整理 算法 | <ul style="list-style-type: none"> 标记阶段：标记出所有需要回收的对象； 整理阶段：让所有存活的对象都向一端移动； 清除阶段：统一清除（回收）端以外的对象。 | <ul style="list-style-type: none"> 解决了标记-清除算法中 清除效率低的问题：一次清除端外区域。 解决了标记-清除算法中 空间产生不连续内存碎片的问题：将已使用内存上的存活对象 移动到栈顶的指针，按顺序分配内存即可。 | 步骤繁多：标记、整理、清除 | 对象存活率较低 & 垃圾回收行为频率低 <small>(如老年代)</small> |
| 分代收集 算法 | <ul style="list-style-type: none"> 根据对象存活周期的不同将 Java 堆内存分为：新生代 & 老年代 每块区域特点如下： <ul style="list-style-type: none"> 新生代：对象存活率较低 & 垃圾回收行为频率高 老年代：对象存活率较低 & 垃圾回收行为频率低 根据每块区域特点选择对应的垃圾收集算法（即上面介绍的算法）： <ul style="list-style-type: none"> 新生代：采用 复制算法 老年代：采用 标记-清除 算法、标记 - 整理 算法 | 效率高、空间利用高：根据不同区域特点选择不同垃圾收集算法 | | 虚拟机基本都采用这种算法 |

2.2 常见的内存问题 & 优化方案

- 常见的内存问题如下

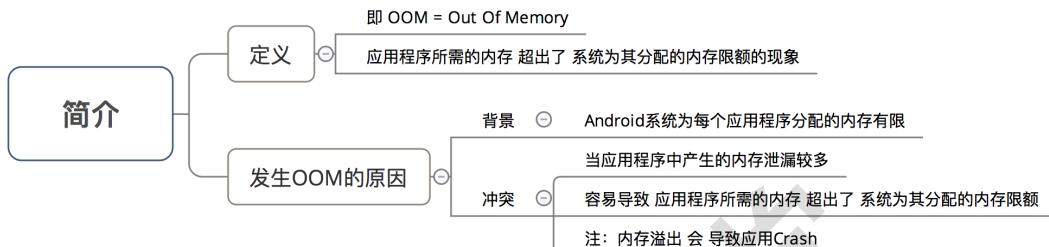
- 内存泄露
- 内存抖动
- 图片 `Bitmap` 相关
- 代码质量 & 数量
- 日常不正确使用

- 下面，我将详细分析每项的内存问题 & 给出优化方案

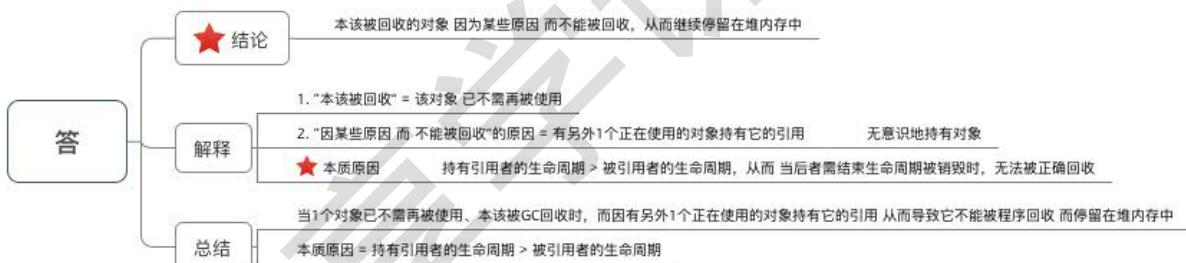
2.2.1 内存泄露

- 简介
即 **ML** (Memory Leak)，指 程序在申请内存后，当该内存不需再使用 **但却无法被释放 & 归还给 程序**的现象
- 对应用程序的影响
容易使得应用程序发生内存溢出，即 **oom**

内存溢出 简介：



- 发生内存泄露的本质原因



- 常见内存泄露原因
 1. 集合类
 2. **static**关键字修饰的成员变量
 3. 非静态内部类 / 匿名类
 4. 资源对象使用后未关闭
- 优化方案
具体请看文章：[Android性能优化：手把手带你全面了解 内存泄露 & 解决方案](#)
- ○ *

2.2.2 图片资源Bitmap相关

- 优化原因

即 为什么要优化图片 Bitmap 资源，具体如下图：



- 优化方向

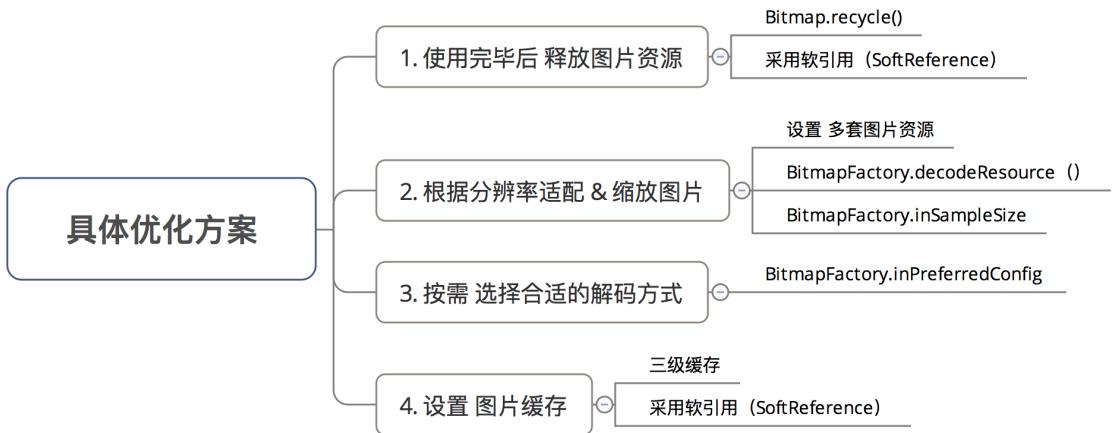
主要从以下方面优化图片 Bitmap 资源的使用 & 内存管理



- 具体优化方案

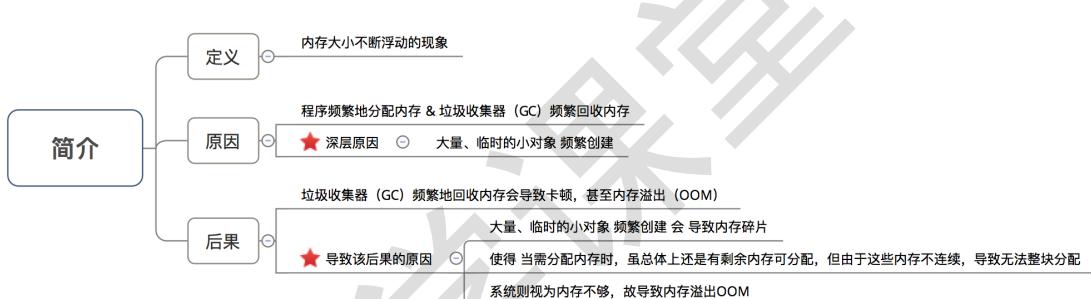
下面，我将详细讲解每个优化方向的具体优化方案

关于更加具体的介绍，请看文章：[Android性能优化：那些关于Bitmap优化的小事](#)



2.2.3 内存抖动

- 简介



- 优化方案

尽量避免频繁创建大量、临时的小对象

- ○ *

2.2.4 代码质量 & 数量

- 优化原因

代码本身的质量（如 数据结构、数据类型等） & 数量（代码量的大小）可能会导致大量的内存问题，如占用内存大、内存利用率低等

• 优化方案

主要从代码总量、数据结构、数据类型、& 数据对象引用方面优化，具体如下

| 优化方向 | 优化方案 | 具体描述 | 备注 |
|--------|---|--|--|
| 代码数据总量 | 不必要的类、对象 & 功能库会带来巨大内存开销 • 如 每个类(含 匿名内部类)约 使用500个字节 • 每个类实例在运行内存 (RAM) 中占用12 – 16个字节 | <ul style="list-style-type: none"> 减少不必要的类 & 对象 减少引入不必要的库 使用代码混淆 | 代码混淆的作用：去除无用的代码 & 通过语义模糊 重命名类、字段 & 方法从而缩小、优化代码，从而使得代码更简洁、更少量的RAM映射页 |
| | 尽可能减少 数据体积大小 | 通过序列化数据 从而减少数据体积大小 (如 使用Google出品的ProtocolBuffer，可节省30%的数据大小) | <ul style="list-style-type: none"> 慎用SharePreference 因对于同一个sp，会将整个xml文件载入内存，容易出现：为了读1个配置，就将几百K的数据读进内存 |
| 数据结构 | 使用性能高的数据结构 | <ul style="list-style-type: none"> 利用Android优化后的数据容器 取代传统的HashMap 如：SparseArray, SparseBooleanArray, LongSparseArray 使用SparseArray代替 key为int的HashMap，可省30%的内存 ArrayMap使用代替 key为其他类型的HashMap，可省10%的内存 | <ul style="list-style-type: none"> ArrayMap 和 SparseArray是Android的系统API 专门为移动设备定制，用于取代HashMap 而达到节省内存的目的 注： a. 传统的HashMap在内存上的实现十分低效的原因 = 需为map中每项在内存中建立映射关系 b. SparseArray更高效的原因 = 避免系统中需自动封箱(autobox)的key |
| 数据类型 | 使用占用内存小的数据类型 | 尽量避免使用枚举类型 | 因为枚举变量占用内存大，比直接使用int类型多使用2倍内存 |
| 数据对象引用 | 根据不同的应用场景，选择不同的引用类型 (强、软、弱、虚) | <ul style="list-style-type: none"> 强引用：该变量不希望被垃圾回收器回收 软引用：缓存机制（即 实现内存敏感的数据缓存，如图片、网页缓存等） 弱引用：防止内存泄漏、保护对象引用 虚引用：跟踪对象被垃圾回收器回收的活动 | |

● ○ *

4.5 常见使用

• 优化原因

一些常见使用也可能引发大量的内存问题，下面我将详细介绍。

• 优化方案

| 优化方向 | 具体描述 | 解决方案 | 备注 |
|------------------|---|---|--|
| Adapter、ListView | <ul style="list-style-type: none"> 在滑动ListView获取最新的View时，容易频繁生成大量对象 即 每次都在getView()中重新实例化一个View对象 不仅浪费资源、时间，也将使得内存占用越来越大，从而使得内存泄露 | <ul style="list-style-type: none"> ListView复用、缓存优化，具体如下 使用缓存的convertView 直接使用ViewHolder | 1. 初始时，ListView会根据当前的屏幕布局 从Adapter中实例化一定数量的View对象，同时ListView会将这些View对象缓存起来 2. 当向上、下滚动ListView时，原先位于最上、下面的ListView的View对象会被回收，然后被用来构造新出现的最下面的list item a. 这个构造过程由getView()方法完成：来的ListView提供每一个item所需要的view对象 b. getView()的第2个参数为 convertView – 被缓存起来的list item的view对象 c. 初始时缓存中没有view对象，则convertView = null |
| 服务Service | <ul style="list-style-type: none"> 当启动常驻服务Service时，系统会优先保持服务在后台不断运行 服务Service使用的内存不能做其它事情 & 会减少系统的LRU缓存处理数目 最终导致App使用、切换效率降低，从而导致内存应用效率低 | <ul style="list-style-type: none"> 尽量减少使用常驻服务Service 尽量使用 IntentService 控制 Service 的生命周期 (当Service执行完所有的任务时 (intent) 会自动停止) | |
| 依赖注入框架 | <ul style="list-style-type: none"> 框架的运行方式：通过注解的方式扫描代码，从而执行一系列的初始化 该运行方式把一些我们不需要的大对象映射到内存中 被映射后的数据会被分配到干净的内存中，很长一段时间都不会使用，从而造成了内存大量浪费 | 避免使用 依赖注入框架 | 依赖注入框架的作用：通过简单代码 & 自适应环境 即可进行有用的测试 和其它配置的更改 |
| 多进程 | 进程十分占用内存 (空进程也会占用内存 = 1M) | 尽量少用多进程 | 为了保活 & 提高稳定性，现在很多App都会进行进程拆分 = 多进程 |

注：

- 还有1个内存优化的终极方案：调大 虚拟机Dalvik的堆内存大小
- 即 在 `AndroidManifest.xml` 的 `application` 标签中增加一个 `android:largeHeap` 属性 (值 = `true`)，从而通知虚拟机 应用程序需要更大的堆内存
- 但不建议 & 不鼓励该做法

4.6 额外小技巧

此处，还有一些内存优化的小技巧希望告诉大家

- 技巧1：获取当前可使用的内存大小

调用 `ActivityManager.getMemoryClass()` 方法可获取当前应用可用的内存大小（单位 = 兆）

- 技巧2：获取当前的内存使用情况

在应用生命周期的任何阶段，调用 `onTrimMemory()` 获取应用程序当前内存使用情况（以内存级别进行识别），可根据该方法返回的**内存紧张级别参数**来释放内存

Android 4.0 后提供的一个API

| 内存使用情况 级别 (级别越高、越严重) | 含义 |
|------------------------------|--|
| TRIM_MEMORY_RUNNING_MODERATE | <ul style="list-style-type: none">• 内存不足级别：5• 状态：应用程序处于前台运行• 含义：应用程序正常运行，不会被杀掉。但当前内存有点低，系统开始杀死其他进程后台应用程序 |
| TRIM_MEMORY_RUNNING_LOW | <ul style="list-style-type: none">• 内存不足级别：10• 状态：应用程序处于前台运行• 含义：应用程序正常运行，不会被杀掉。但当前内存非常低了，请必须释放你自身不必要的内存，否则会影响应用程序的性能（如响应速度等等） |
| TRIM_MEMORY_RUNNING_CRITICAL | <ul style="list-style-type: none">• 内存不足级别：15• 状态：应用程序处于前台运行• 含义：应用程序正常运行，但大部分其他后台程序已被杀死，请务必释放你自身不必要的内存，否则你也会被杀死 |
| TRIM_MEMORY_UI_HIDDEN | <ul style="list-style-type: none">• 内存不足级别：20• 状态：应用程序处于前台运行• 含义：系统内存已经非常低了，并将该应用程序从前台切换到后台，即收回UI资源 |
| TRIM_MEMORY_BACKGROUND | <ul style="list-style-type: none">• 内存不足级别：40• 状态：应用程序处于后台缓存• 含义：系统内存已经降低了，该应用程序处于LRU缓存列表的最近位置，但不会被清理掉。• 此时应该释放掉一些较容易恢复的资源让手机的内存变得充足，从而让我们的程序更长时间地保留在缓存当中 |
| TRIM_MEMORY_MODERATE | <ul style="list-style-type: none">• 内存不足级别：60• 状态：应用程序处于后台缓存• 含义：系统内存已经非常低了，该应用程序处于LRU缓存列表的中间位置，若手机内存再得不到释放，该应用程序有被系统杀死的风险 |
| TRIM_MEMORY_COMPLETE | <ul style="list-style-type: none">• 内存不足级别：80• 状态：应用程序处于后台缓存• 含义：内存严重不足，该应用程序已处于LRU缓存列表的最边缘位置，应用程序随时都有被回收的风险，此时应该把一切可以释放的资源都释放从而避免被杀死 |

- 技巧3：当视图变为隐藏状态时，则释放内存

当用户跳转到不同的应用 & 视图不再显示时，应释放应用视图所占的资源

1. 注：此时释放所占用的资源能显著的提高系统的缓存处理容量

2. 具体操作：实现当前Activity类的onTrimMemory()后，当用户离开视图时会得到通知；若得到返回的参数 = TRIM_MEMORY_UI_HIDDEN 即代表视图变为隐藏状态，则可释放视图所占用的资源。

2.3 辅助内存优化的分析工具

- 哪怕完全了解内存的原因，但难免还是会发现人为难以发现的内存问题
- 下面将简单介绍几个主流的辅助分析内存优化的工具，分别是
 1. MAT(Memory Analysis Tools)
 2. Heap Viewer
 3. Allocation Tracker
 4. Android Studio 的 Memory Monitor
 5. LeakCanary

2.3.1 MAT(Memory Analysis Tools)

- 定义：一个Eclipse的Java Heap 内存分析工具 ->[下载地址](#)
- 作用：查看当前内存占用情况

通过分析Java进程的内存快照 HPROF 分析，快速计算出在内存中对象占用的大小，查看哪些对象不能被垃圾收集器回收 & 可通过视图直观地查看可能造成这种结果的对象

- 具体使用：[MAT使用攻略](#)

2.3.2 Heap Viewer

- 定义：一个的 Java Heap 内存分析工具
- 作用：查看当前内存快照

可查看分别有哪些类型的数据在堆内存总 & 各种类型数据的占比情况

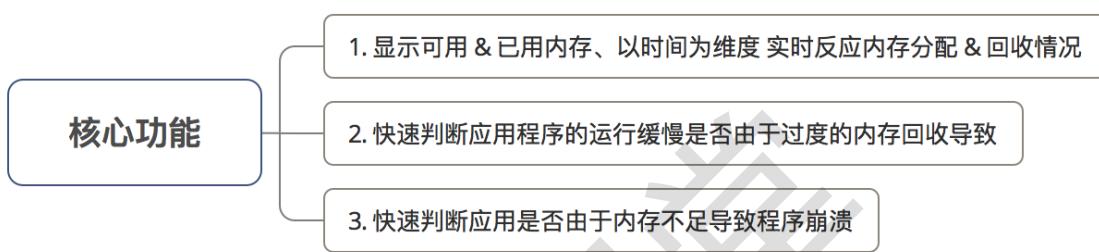
- 具体使用：[Heap Viewer使用攻略](#)

2.3.3 Allocation Tracker

- 简介：一个内存追踪分析工具
- 作用：追踪内存分配信息，按顺序排列
- 具体使用：[Allocation Tracker使用攻略](#)

2.3.4 Memory Monitor

- 简介：一个 `Android Studio` 自带的图形化检测内存工具
- 作用：跟踪系统 / 应用的内存使用情况。核心功能如下



- 具体使用：[Android Studio 的 Memory Monitor使用攻略](#)

2.3.5 LeakCanary

- 简介：一个 `square` 出品的 `Android` 开源库 ->>[下载地址](#)
- 作用：检测内存泄露
- 具体使用：<https://www.liaohuqiu.net/cn/posts/leak-canary/>

至此，关于内存优化的所有知识讲解完毕

2.3.6 总结

- 本文主要讲解内存优化的相关知识，总结如下：

| 优化方向 | 优化原因 | 优化方案 |
|--------------|--|---|
| 内存泄露 | 程序在申请内存后，当该内存不再使用但却无法被释放 & 归还给程序的现象，从而使得应用程序发生内存溢出，即 OOM | <p>避免内存泄露，分别从以下方面入手：</p> <ul style="list-style-type: none"> 集合类：回收集合元素 Static关键字修饰的成员变量：避免static引用过多实例 非静态内部类 / 匿名类：使用静态内部类 资源对象使用后未关闭：关闭资源对象 |
| 内存抖动 | 应用程序频繁分配内存 & 垃圾回收器（GC）频繁回收内存容易导致卡顿，甚至出现内存溢出（OOM） | 尽量避免频繁大量、临时的小对象 |
| 图片Bitmap相关 | 图片Bitmap资源 占用App的大部分内存 & 容易出现使用不当、内存缺乏管理现象 | <ul style="list-style-type: none"> 按需 选择合适的解码方式 设置 图片缓存 根据分辨率适配 & 缩放图片 使用完毕后 释放图片资源 |
| 代码质量 & 数量 | 代码本身的质量（如 数据结构、数据类型等）& 数量（代码量的大小）可能会导致大量的内存问题 | <ul style="list-style-type: none"> 减少不必要的类、对象、库 & 使用代码混淆 尽可能减少 数据体积大小：序列化数据 使用性能高的数据结构 使用占用内存小的数据类型：避免使用枚举类型 根据不同的应用场景，选择不同的引用类型 |
| 常见使用 | 一些常见使用也可能引发大量的内存问题 | <ul style="list-style-type: none"> ListView复用、缓存 尽量减少使用常驻服务Service 避免使用 依赖注入框架 尽量少用多进程 |
| 优化小技巧 | 通过一些小技能有效提高内存优化效率 | <ul style="list-style-type: none"> 获取当前可使用的内存大小 获取当前的内存使用情况 当视图变为隐藏状态时，则释放内存 |
| 使用 辅助内存优化的工具 | 人为难以发现隐蔽的内存问题，通过工具辅助分析 | <ul style="list-style-type: none"> MAT(Memory Analysis Tools) Heap Viewer Allocation Tracker Memory Monitor LeakCanary |

2.4 Android性能优化：手把手带你全面了解 内存泄露 & 解决方案

前言

- 在Android中，内存泄露的现象十分常见；而内存泄露导致的后果会使得应用Crash
- 本文全面介绍了内存泄露的本质、原因 & 解决方案，最终提供一些常见的内存泄露分析工具，希望你们会喜欢。

2.4.1简介

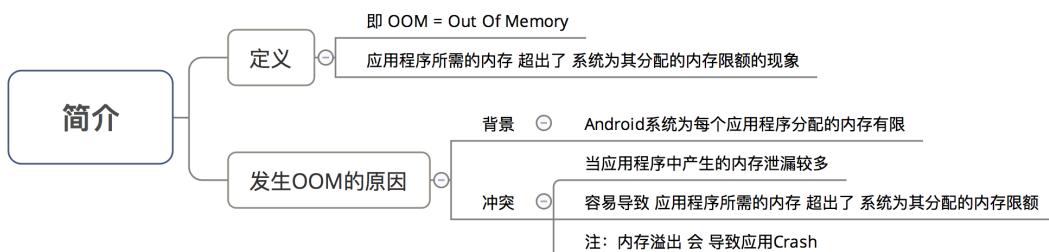
- 即 ML (Memory Leak)
- 指程序在申请内存后，当该内存不再使用但却无法被释放 & 归还给程序的现象

- ○ *

2.4.2 对应用程序的影响

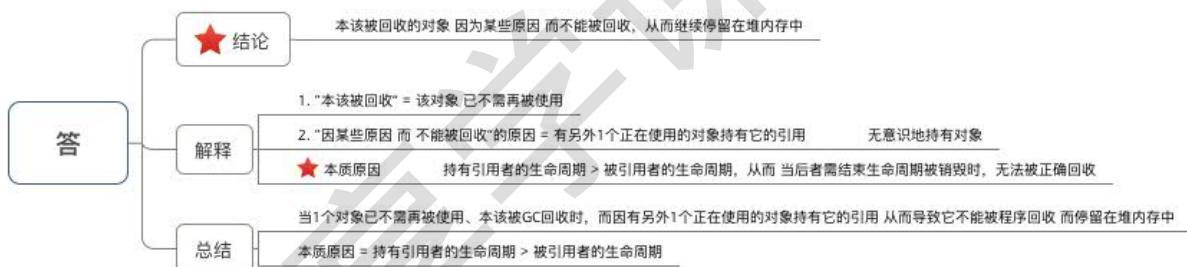
- 容易使得应用程序发生内存溢出，即 OOM

内存溢出 简介：



2.4.3 发生内存泄露的本质原因

- 具体描述



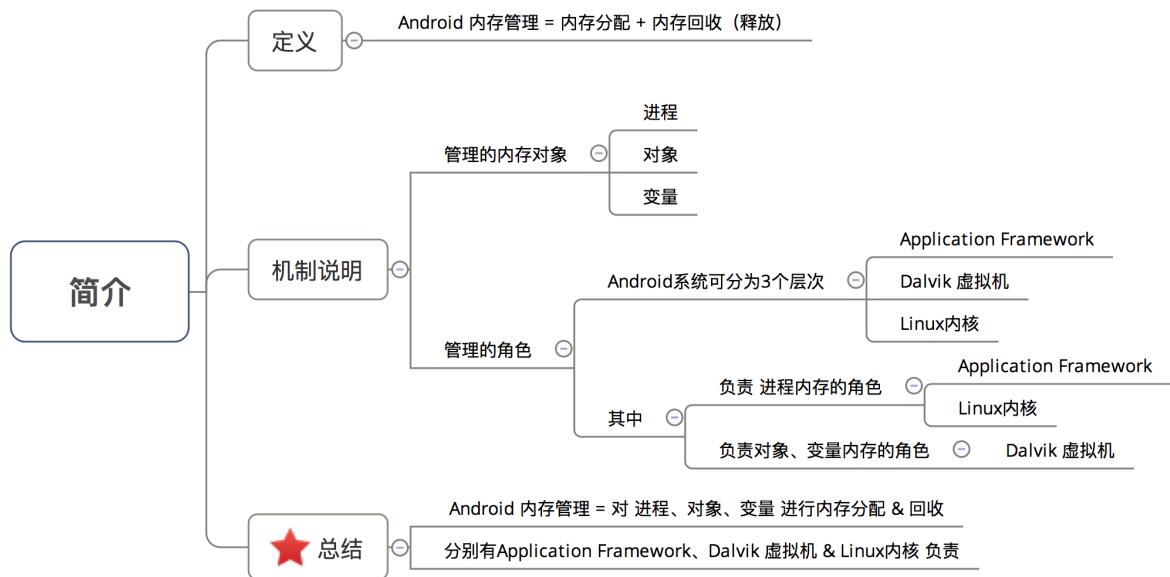
- 特别注意

从机制上的角度来说，由于 Java 存在垃圾回收机制（GC），理应不存在内存泄露；出现内存泄露的原因仅仅是外部人为原因 = **无意识地持有对象引用，使得 持有引用者的生命周期 > 被引用者的生命周期**

- ○ *

2.4.4 储备知识：Android 内存管理机制

4.1 简介



下面，将针对回收 进程、对象、变量的内存分配 & 回收进行详细讲解

4.2 针对进程的内存策略

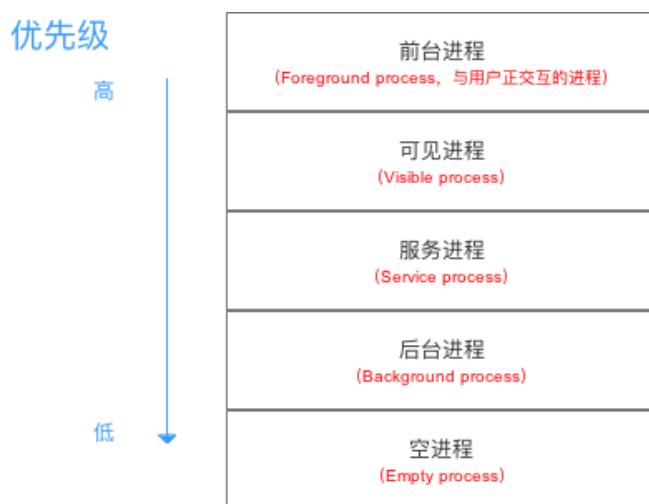
a. 内存分配策略

由 `ActivityManagerService` 集中管理 所有进程的内存分配

b. 内存回收策略

- 步骤1： `Application Framework` 决定回收的进程类型
Android中的进程是托管的；当进程空间紧张时，会按进程优先级低->高的顺序自动回收进程

Android将进程分为5个优先等级，具体如下：



注：当系统需回收进程时，最先回收空进程、最后回收前台进程

- 步骤2： `Linux` 内核真正回收具体进程

1. `ActivityManagerService` 对所有进程进行评分（评分存放在变量 `adj` 中）
2. 更新评分到 `Linux` 内核
3. 由 `Linux` 内核完成真正的内存回收

此处仅总结流程，这其中的过程复杂，有兴趣的读者可研究系统源码

`ActivityManagerService.java`

4.3 针对对象、变量的内存策略

- `Android` 的对于对象、变量的内存策略同 `Java`
- 内存管理 = 对象 / 变量的内存分配 + 内存释放

下面，将详细讲解 **内存分配 & 内存释放策略**

a. 内存分配策略

- 对象 / 变量的内存分配 **由程序自动负责**
- 共有3种：静态分配、栈式分配、& 堆式分配，分别面向静态变量、局部变量 & 对象实例
- 具体介绍如下

| 内存分配策略 | 使用的内存空间 | 存储的数据 | 分配策略 描述 |
|---------------------------------|-------------------------------|--|---|
| 静态分配 | 方法区 <small>(静态存储区)</small> | 存储 已被虚拟机加载的类信息、常量、静态变量 | 在程序编译时就已分配好 & 存在于程序整个运行期间 <small>(不需回收)</small> |
| 栈式分配 | 栈区 <small>(Stack)</small> | 存储方法执行时的局部变量（含数据类型、对象的引用） <small>(以帧的形式)</small> | <ul style="list-style-type: none"> 方法执行时，定义局部变量 则由程序自动在栈中分配内存 方法执行结束 / 超出变量域时，则由帧栈自动释放该部分内存 效率高 <small>(因 栈内存分配运算内置于处理器的指令集中)</small> 但分配的内存容量有限 |
| 堆式分配 <small>(动态内存分配)</small> | 堆区 <small>(Heap)</small> | 存储Java对象的实例 & 实例内的成员变量 <ul style="list-style-type: none"> 即 采用关键字 <code>new</code> 出来的对象 实例的成员变量 = 基本数据类型、引用 & 引用的对象实体 | <ul style="list-style-type: none"> 创建对象实例时，由程序分配 <small>(由Java垃圾回收管理器 自动管理；不使用时则回收)</small> 访问方式 <ol style="list-style-type: none"> 在堆中创建1个对象 / 数组 & 在栈中定义一个特殊的变量（引用变量） = 数组 / 对象在堆内存中的首地址 通过 引用变量 来访问堆内存中的对象 / 数组 |

注：用1个实例讲解 内存分配

```
public class Sample {
    int s1 = 0;
    Sample mSample1 = new Sample();

    // 方法中的局部变量s2、mSample2存放在 栈内存
    // 变量mSample2所指向的对象实例存放在 堆内存
    // 该实例的成员变量s1、mSample1也存放在栈中
```

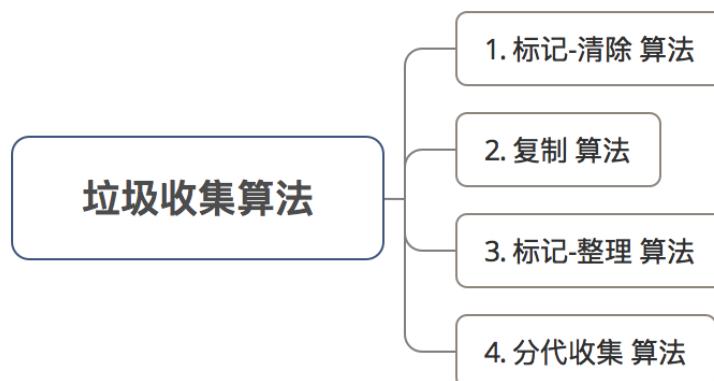
```
public void method() {  
    int s2 = 0;  
    Sample mSample2 = new Sample();  
}  
}  
  
// 变量mSample3所指向的对象实例存放在堆内存  
// 该实例的成员变量s1、mSample1也存放在栈中  
Sample mSample3 = new Sample();
```

b. 内存释放策略

- 对象 / 变量的内存释放由 Java 垃圾回收器 (GC) / 帧栈负责
- 此处主要讲解对象分配（即堆式分配）的内存释放策略 = Java 垃圾回收器 (GC)

由于静态分配不需释放、栈式分配仅通过帧栈自动出、入栈，较简单，故不详细描述

- Java 垃圾回收器 (GC) 的内存释放 = 垃圾回收算法，主要包括：



- 具体介绍如下

| 算法名称 | 算法思想 | 优点 | 缺点 | 应用场景 |
|------------|--|------------------------------|--|--|
| 标记-清除 算法 | <ul style="list-style-type: none"> 标记阶段：标记出所有需要回收的对象； 清除阶段：统一清除（回收）所有被标记的对象。 | 实现简单 | <ul style="list-style-type: none"> 效率问题：标记和清除 两个过程效率不高 空间问题：标记 - 清除后，会产生大量不连续的内存碎片。 | 对象存活率较低 & 垃圾回收行为频率低 <small>(如老年代)</small> |
| 复制 算法 | <ul style="list-style-type: none"> 将内存分为大小相等的两块，每次使用其中一块； 当 使用的这块内存用完，就将 这块内存上还存活的对象 复制到另一块还没有用过的内存上； 最终将使用的那块内存一次清理掉。 | | <ul style="list-style-type: none"> 解决了标记-清除算法中 清除效率低的问题：每次仅回收内存的一半区域 解决了标记-清除算法中 空间产生不连续内存碎片的问题：将已使用内存上的存活对象 移动到栈顶的指针，按顺序分配内存即可。 | 对象存活率较低 & 需要频繁进行垃圾回收 的区域 <small>(如新生儿代)</small> |
| 标记 - 整理 算法 | <ul style="list-style-type: none"> 标记阶段：标记出所有需要回收的对象； 整理阶段：让所有存活的对象都向一端移动； 清除阶段：统一清除（回收）端以外的对象。 | | <ul style="list-style-type: none"> 解决了标记-清除算法中 清除效率低的问题：一次清除端外区域 解决了标记-清除算法中 空间产生不连续内存碎片的问题：将已使用内存上的存活对象 移动到栈顶的指针，按顺序分配内存即可。 | 对象存活率较低 & 垃圾回收行为频率低 <small>(如老年代)</small> |
| 分代收集 算法 | <ul style="list-style-type: none"> 根据对象存活周期的不同将 Java堆内存分为：新生代 & 老年代 每块区域特点如下： <ul style="list-style-type: none"> 新生代：对象存活率较低 & 垃圾回收行为频率高 老年代：对象存活率较高 & 垃圾回收行为频率低 根据每块区域特点选择对应的垃圾收集算法（即上面介绍的算法）： <ul style="list-style-type: none"> 新生代：采用 复制算法 老年代：采用 标记-清除 算法、标记 - 整理 算法 | 效率高、空间利用高：根据不同区域特点选择不同垃圾收集算法 | | 虚拟机基本都采用这种算法 |

2.4.5 常见的内存泄露原因 & 解决方案

- 常见引发内存泄露原因主要有：

1. 集合类
2. `static`关键字修饰的成员变量
3. 非静态内部类 / 匿名类
4. 资源对象使用后未关闭

- 下面，我将详细介绍每个引发内存泄露的原因

5.1 集合类

- 内存泄露原因

集合类 添加元素后，仍引用着 集合元素对象，导致该集合元素对象不可被回收，从而 导致内存泄漏

- 实例演示

```
// 通过 循环申请Object 对象 & 将申请的对象逐个放入到集合List
List<Object> objectList = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    Object o = new Object();
    objectList.add(o);
    o = null;
}
// 虽释放了集合元素引用的本身: o=null
// 但集合List 仍然引用该对象，故垃圾回收器GC 依然不可回收该对象
```

- 解决方案

集合类 添加集合元素对象 后，在使用后必须从集合中删除

由于1个集合中有许多元素，故最简单的方法 = 清空集合对象 & 设置为null

```
// 释放objectList  
objectList.clear();  
objectList=null;
```

5.2 Static 关键字修饰的成员变量

- 储备知识

被 `static` 关键字修饰的成员变量的生命周期 = 应用程序的生命周期

- 泄露原因

若使被 `static` 关键字修饰的成员变量 引用耗费资源过多的实例（如 `Context`），则容易出现该成员变量的生命周期 > 引用实例生命周期的情况，当引用实例需结束生命周期销毁时，会因静态变量的持有而无法被回收，从而出现内存泄露

- 实例讲解

```
public class className {  
    // 定义1个静态变量  
    private static Context mContext;  
    //...  
    // 引用的是Activity的context  
    mContext = context;  
  
    // 当Activity需销毁时，由于mContext = 静态 & 生命周期 = 应用程序的生命周期，故 Activity无法被回收，从而出现内存泄露  
  
}
```

- 解决方案

1. 尽量避免 `static` 成员变量引用资源耗费过多的实例（如 `Context`）

| 若需引用 `Context`, 则尽量使用 `Applicaiton` 的 `Context`

2. 使用 弱引用 (`weakReference`) 代替 强引用 持有实例

注：静态成员变量有个非常典型的例子 = 单例模式

- 储备知识

单例模式 由于其**静态特性**, 其生命周期的长度 = 应用程序的生命周期

- 泄露原因

若1个对象已不需再使用 而单例对象还持有该对象的引用, 那么该对象将不能被正常回收 从而 导致内存泄漏

- 实例演示

```
// 创建单例时, 需传入一个Context
// 若传入的是Activity的Context, 此时单例 则持有该Activity的引用
// 由于单例一直持有该Activity的引用 (直到整个应用生命周期结束), 即使该Activity退出, 该Activity的内存也不会被回收
// 特别是一些庞大的Activity, 此处非常容易导致OOM

public class SingleInstanceClass {
    private static SingleInstanceClass instance;
    private Context mContext;
    private SingleInstanceClass(Context context) {

        this.mContext = context; // 传递的是Activity的
context
    }

    public SingleInstanceClass getInstance(Context
context) {
        if (instance == null) {
            instance = new SingleInstanceClass(context);
        }
        return instance;
    }
}
```

- 解决方案

单例模式引用的对象的生命周期 = 应用的生命周期

如上述实例，应传递 Application 的 Context，因 Application 的生命周期 = 整个应用的生命周期

```
public class SingleInstanceClass {  
    private static SingleInstanceClass instance;  
    private Context mContext;  
    private SingleInstanceClass(Context context) {  
  
        this.mContext = context.getApplicationContext();  
        // 传递的是Application 的context  
    }  
  
    public SingleInstanceClass getInstance(Context context) {  
        if (instance == null) {  
            instance = new SingleInstanceClass(context);  
        }  
        return instance;  
    }  
}
```

5.3 非静态内部类 / 匿名类

- 储备知识

非静态内部类 / 匿名类 默认持有 外部类的引用；而静态内部类则不会

- 常见情况

3种，分别是：非静态内部类的实例 = 静态、多线程、消息传递机制
(Handler)

5.3.1 非静态内部类的实例 = 静态

- 泄露原因

若 非静态内部类所创建的实例 = 静态 (其生命周期 = 应用的生命周期)

期)，会因**非静态内部类默认持有外部类的引用**而导致外部类无法释放，最终造成内存泄露

即外部类中持有非静态内部类的静态对象

- 实例演示

```
// 背景:
```

- a. 在启动频繁的Activity中，为了避免重复创建相同的数据资源，会在Activity内部创建一个非静态内部类的单例
- b. 每次启动Activity时都会使用该单例的数据

```
public class TestActivity extends AppCompatActivity {
```

```
    // 非静态内部类的实例的引用
```

```
    // 注：设置为静态
```

```
    public static InnerClass innerClass = null;
```

```
    @Override
```

```
    protected void onCreate(@Nullable Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        // 保证非静态内部类的实例只有1个
```

```
        if (innerClass == null)
```

```
            innerClass = new InnerClass();
```

```
}
```

```
    // 非静态内部类的定义
```

```
    private class InnerClass {
```

```
        //...
```

```
}
```

```
}
```

```
// 造成内存泄露的原因：
```

```
    // a. 当TestActivity销毁时，因非静态内部类单例的引用  
    // (innerClass) 的生命周期 = 应用App的生命周期、持有外部类  
    // TestActivity的引用
```

```
// b. 故 TestActivity无法被GC回收，从而导致内存泄漏
```

- 解决方案

1. 将非静态内部类设置为：静态内部类（静态内部类默认不持有外部类的引用）
2. 该内部类抽取出来封装成一个单例
3. 尽量避免非静态内部类所创建的实例 = 静态

| 若需使用 Context，建议使用 Application 的 Context

5.3.2 多线程： AsyncTask、实现Runnable接口、继承Thread类

- 储备知识

多线程的使用方法 = 非静态内部类 / 匿名类；即 线程类 属于 非静态内部类 / 匿名类

- 泄露原因

当 工作线程正在处理任务 & 外部类需销毁时，**由于 工作线程实例 持有外部类引用**，将使得外部类无法被垃圾回收器（GC）回收，从而造成 内存泄露

| 1. 多线程主要使用的是： AsyncTask、实现 Runnable 接口 & 继承 Thread 类

| 2. 前3者内存泄露的原理相同，此处主要以继承 Thread 类 为例说明

- 实例演示

```
/**  
 * 方式1：新建Thread子类（内部类）  
 */  
public class MainActivity extends  
AppCompatActivity {  
  
    public static final String TAG = "carson: ";  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```

```
// 通过创建的内部类 实现多线程
new MyThread().start();

}

// 自定义的Thread子类
private class MyThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep(5000);
            Log.d(TAG, "执行了多线程");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

/**
 * 方式2：匿名Thread内部类
 */
public class MainActivity extends AppCompatActivity
{

    public static final String TAG = "carson: ";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 通过匿名内部类 实现多线程
        new Thread() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                }
            }
        }.start();
    }
}
```

```

        Log.d(TAG, "执行了多线程");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}
}.start();
}

}

/**
 * 分析：内存泄露原因
 */
// 工作线程Thread类属于非静态内部类 / 匿名内部类，运行时默认持有
// 外部类的引用
// 当工作线程运行时，若外部类MainActivity需销毁
// 由于此时工作线程类实例持有外部类的引用，将使得外部类无法被垃圾
// 回收器（GC）回收，从而造成 内存泄露

```

- 解决方案

从上面可看出，造成内存泄露的原因有2个关键条件：

1. 存在“工作线程实例 持有外部类引用”的引用关系
2. 工作线程实例的生命周期 > 外部类的生命周期，即工作线程仍在运行而 外部类需销毁

解决方案的思路 = 使得上述任1条件不成立 即可。

```
// 共有2个解决方案：静态内部类 & 当外部类结束生命周期时，强制结束线程
```

```
// 具体描述如下
```

```
/**
```

- * 解决方式1：静态内部类

- * 原理：静态内部类 不默认持有外部类的引用，从而使得“工作线程实例 持有 外部类引用”的引用关系 不复存在

- * 具体实现：将Thread的子类设置成 静态内部类

```
 */
public class MainActivity extends
AppCompatActivity {

    public static final String TAG = "carson: ";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 通过创建的内部类 实现多线程
        new MyThread().start();

    }
    // 分析1：自定义Thread子类
    // 设置为：静态内部类
    private static class MyThread extends Thread{
        @Override
        public void run() {
            try {
                Thread.sleep(5000);
                Log.d(TAG, "执行了多线程");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * 解决方案2：当外部类结束生命周期时，强制结束线程
 * 原理：使得 工作线程实例的生命周期 与 外部类的生命周期 同步
 * 具体实现：当 外部类（此处以Activity为例） 结束生命周期时
（此时系统会调用onDestroy（）），强制结束线程（调用stop（））
*/
@Override
protected void onDestroy() {
    super.onDestroy();
```

```
    Thread.stop();
    // 外部类Activity生命周期结束时，强制结束线程
}
```

5.3.3 消息传递机制：Handler

具体请看文章：[Android 内存泄露：详解 Handler 内存泄露的原因](#)

5.4 资源对象使用后未关闭

- **泄露原因**

对于资源的使用（如广播BroadcastReceiver、文件流File、数据库游标Cursor、图片资源Bitmap等），若在Activity销毁时无及时关闭 / 注销这些资源，则这些资源将不会被回收，从而造成内存泄漏

- **解决方案**

在Activity销毁时 及时关闭 / 注销资源

```
// 对于 广播BroadcastReceiver：注销注册
unregisterReceiver()

// 对于 文件流File：关闭流
InputStream / OutputStream.close()

// 对于数据库游标cursor： 使用后关闭游标
cursor.close()

// 对于 图片资源Bitmap： Android分配给图片的内存只有8M，若1个
Bitmap对象占内存较多，当它不再被使用时，应调用recycle()回收此对象
的像素所占用的内存；最后再赋为null
Bitmap.recycle();
Bitmap = null;

// 对于动画（属性动画）
// 将动画设置成无限循环播放repeatCount = “infinite”后
// 在Activity退出时记得停止动画
```

5.5 其他使用

- 除了上述4种常见情况，还有一些日常的使用会导致内存泄露
- 主要包括：Context、WebView、Adapter，具体介绍如下

| 原因 | 具体描述 | 解决方案 | 备注 |
|---------|---|---|--|
| Context | 当拥有该Activity Context参数对象仍在使用、而该Activity需销毁时，该Activity则由于被保持引用而无法被回收，从而造成内存泄露 <small>(当拥有Context参数的对象的生命周期 > 该Context参数的生命周期时，则容易出现内存泄露)</small> | 对Context的引用不要超过它本身的生命周期 • 如：尽量使用ApplicationContext代替ActivityContext • 因ApplicationContext会随着应用程序的存在而存在，而不依赖于activity的生命周期 | • 在Android中，通常可使用Context对象有2种：Activity、Application • 当类 / 方法需Context对象时，常优先使用 Activity作为Context参数；此时该对象对整个activity保持引用 |
| WebView | 不再使用WebView对象后 无销毁，导致占用的内存长期无法被回收，从而造成内存泄露 | 通过多线程在不使用WebView对象时进行销毁 | 如： 1. 为WebView开启另一个进程 2. 通过ADL与主线程进行通信，WebView所在的进程可根据业务的需要，选择在合适的时机销毁，从而达到内存的完整释放 |
| Adapter | • 在滑动ListView获取最新的View时，容易频繁生成大量对象 • 即 每次都在getView()中重新实例化1个View对象 • 不仅浪费资源、时间，也将使得内存占用越来越大，从而使得内存泄露 | • 使用缓存的convertView • 直接使用 ViewHolder | 1. 初始时，ListView会根据当前的屏幕布局从Adapter中实例化一定数量的View对象，同时ListView会将这些View对象缓存起来 2. 当向上、下滚动ListView时，原先位于最上、下面的List Item的View对象会被回收，然后被用来构造新出现的最下面的List Item a. 这个构造过程由getView()方法完成；来向ListView提供每一个Item所需要的View对象 b. getViewHolder()的第2个形参View convertView = 被缓存起来的list_item的view对象 c. 初始化时缓存中没有View对象，则convertView = null |

5.6 总结

下面，我将用一张图总结Android中内存泄露的原因 & 解决方案

| 原因 | 具体描述 | 解决方案 | 备注 |
|------------------|---|--|---|
| 集合类 | 集合类 添加元素后，仍引用着集合元素对象，导致该集合元素对象不可被回收，从而导致内存泄漏 | 集合类 添加集合元素对象后，在使用后必须从集中删除 <small>(由于1个集合中有许多元素，故最简单的方法 = 清空集合对象 & 设置为null)</small> | / |
| Static 关键字修饰成员变量 | • 被 Static 关键字修饰的成员变量的生命周期 = 应用程序的生命周期 • 若使用 Static 关键字修饰的成员变量 引用耗资源过多的实例（如Context），则容易出现该成员变量的生命周期 > 引用实例结束生命周期销毁时，会因静态变量持有的而无法被回收，从而出现内存泄露 | • 尽量避免 Static 成员变量引用资源耗过大的实例如 (Context) • (若需引用 Context，则尽量使用ApplicationContext) • 使用 弱引用 (WeakReference) 代替 强引用 | 单例模式 为其实例例子 <small>(单例模式 由于其静态特性，其生命周期的长度 = 应用的生命周期)</small> |
| 非静态内部类 / 匿名类 | 若 非静态内部类所创建的实例 = 静态（其生命周期 = 应用的生命周期），会因 非静态内部类默认持有外部类的引用 而导致外部类无法释放，最终 造成内存泄露 <small>(即 外部类 中 持有 非静态内部类的静态对象)</small> | • 将非静态内部类设置为：静态内部类（静态内部类默认不持有外部类的引用） • 该内部类抽取出来封装成一个单例 • 尽量 避免 非静态内部类所创建的实例 = 静态 <small>(若需使用Context，建议使用 Application 的 Context)</small> | / |
| | 多线程 | 当 工作线程正在处理任务 & 外部类需销毁时，由于 工作线程实例 持有外部类引用，将使得外部类无法被垃圾回收器 (GC) 回收。从而造成 内存泄露 | 线程类 = 静态内部类 • 当外部类结束生命周期时，强制结束线程 |
| 资源对象使用后未关闭 | 当 Handler消息队列 还有未处理的消息 / 正在处理消息时，存在引用关系：“未被处理 / 正处理的消息 -> Handler实例 -> 外部类” • 若出现 Handler的生命周期 > 外部类的生命周期 时 (即 Handler消息队列 还有未处理的消息 / 正在处理消息 而 外部类需销毁时)，将使得外部类无法被垃圾回收器 (GC) 回收，从而造成 内存泄露 | • Handler类 = 静态内部类 + 弱引用持有Activity实例 • 当外部类结束生命周期时，清空Handler内消息队列 | / |
| | 对于资源的使用，若在activity销毁时无及时关闭 / 注销这些资源，则这些资源将不会被回收，从而造成内存泄露 <small>(资源 如 广播BroadcastReceiver、文件File、数据库游标Cursor、图片资源Bitmap等)</small> | 在Activity销毁时 及时关闭 / 注销资源 | / |
| 其他使用 | Context | 当 拥有该Activity Context参数对象仍在使用、而该Activity需销毁时，该Activity则由于被保持引用而无法被回收，从而造成内存泄露 <small>(当拥有Context参数的对象的生命周期 > 该Context参数的生命周期时，则容易出现内存泄露)</small> | 对Context的引用不要超过它本身的生命周期 • 如：尽量使用ApplicationContext代替ActivityContext • 因ApplicationContext会随着应用程序的存在而存在，而不依赖于activity的生命周期 |
| | WebView | 不再使用WebView对象后 无销毁，导致占用的内存长期无法被回收，从而造成内存泄露 | 通过多线程在不使用WebView对象时进行销毁 |
| | Adapter | 在滑动ListView获取最新的View时，每次都在getView()中重新实例化1个View对象。不仅浪费资源、时间，也将使得内存占用越来越大，从而使得内存泄露 | • 使用缓存的convertView • 直接使用 ViewHolder |

2.5 辅助分析内存泄露的工具

- 哪怕完全了解 内存泄露的原因，但难免还是会出现在内存泄露的现象
- 下面将简单介绍几个主流的分析内存泄露的工具，分别是

1. MAT(Memory Analysis Tools)

2. Heap Viewer

3. Allocation Tracker
4. Android Studio 的 Memory Monitor
5. LeakCanary

2.5.1 MAT(Memory Analysis Tools)

- 定义：一个 Eclipse 的 Java Heap 内存分析工具 ->[下载地址](#)
- 作用：查看当前内存占用情况

通过分析 Java 进程的内存快照 HPROF 分析，快速计算出在内存中对象占用的大小，查看哪些对象不能被垃圾收集器回收 & 可通过视图直观地查看可能造成这种结果的对象

- 具体使用：[MAT使用攻略](#)

2.5.2 Heap Viewer

- 定义：一个的 Java Heap 内存分析工具
- 作用：查看当前内存快照

可查看 分别有哪些类型的数据在堆内存总 & 各种类型数据的占比情况

- 具体使用：[Heap Viewer使用攻略](#)

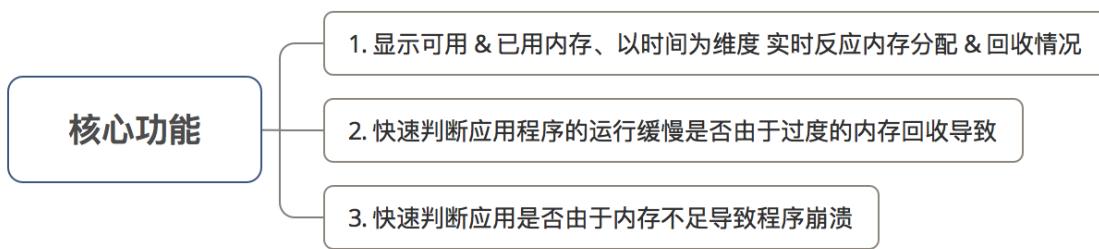
2.5.3 Allocation Tracker

- 简介：一个内存追踪分析工具
- 作用：追踪内存分配信息，按顺序排列
- 具体使用：[Allocation Tracker使用攻略](#)

2.5.4 Memory Monitor

- 简介：一个 Android Studio 自带 的图形化检测内存工具

- 作用：跟踪系统 / 应用的内存使用情况。核心功能如下



- 具体使用：[Android Studio 的 Memory Monitor使用攻略](#)

2.5.5 LeakCanary

- 简介：一个square出品的Android开源库 ->>[下载地址](#)
- 作用：检测内存泄露
- 具体使用：<https://www.liaohuqiu.net/cn/posts/leak-canary/>
- ○ *

2.5.6总结

- 本文全面介绍了内存泄露的本质、原因 & 解决方案，希望大家在开发时尽量避免出现内存泄露

| 原因 | | 具体描述 | 解决方案 | 备注 |
|------------------|--------------|---|--|---|
| 集合类 | | 集合类 添加元素后，仍引用着 集合元素对象，导致该集合元素对象不可被回收，从而 导致内存泄漏 | 集合类 添加集合元素对象后，在使用后必须从集合中删除 (由于1个集合中有许多元素，故最简单的方法 = 清空集合对象 & 设置为null) | / |
| Static 关键字修饰成员变量 | | • 被 Static 关键字修饰的成员变量 的生命周期 = 应用程序的生命周期 • 若使被 Static 关键字修饰的成员变量 引用耗费资源过多的实例 (如Context) ，则容易出现该成员变量的生命周期 > 引用实例生命周期的情况。当引用实例需结束生命周期销毁时，会因静态变量的持有而无法被回收，从而出现内存泄露 | • 尽量避免 Static 成员变量引用资源耗费过多的实例如 (Context) (请弱引用 Context， 则尽量使用Application的Context) • 使用 弱引用 (WeakReference) 替代 强引用 | 单例模式 为其典型例子 (单例模式 由于其静态特性，其生命周期的长度 = 应用的生命周期) |
| 非静态内部类 / 匿名类 | 非静态内部类的实例 静态 | 若 非静态内部类所创建的实例 = 静态 (其生命周期 = 应用的生命周期)：会因 非静态内部类默认持有外部类的引用 而导致外部类无法释放，最终 造成内存泄露 (即 外部类 中 持有 非静态内部类的静态对象) | • 将非静态内部类设置为：静态内部类 (静态内部类默认不持有外部类的引用) • 该内部类抽取出来封装成一个单例 • 尽量 避免 非静态内部类所创建的实例 = 静态 (若需使用Context，建议使用 Application 的 Context) | / |
| | 多线程 | 当 工作线程正在处理任务 & 外部类需销毁时，由于 工作线程实例 持有外部类引用，将使得外部类无法被垃圾回收器 (GC) 回收，从而造成 内存泄露 | • 线程类 = 静态内部类 • 当外部类结束生命周期时，强制结束线程 | 常见的多线程方法 = AsyncTask、实现Runnable接口、继承Thread类 |
| | Handler | • 当Handler消息队列 还有未处理的消息 / 正在处理消息时，存在引用关系：“未被处理 / 正在处理的消息 -> Handler实例 -> 外部类” • 若出现 Handler的生命周期 > 外部类的生命周期 时 (即 Handler消息队列 还有未处理的消息 / 正在处理消息 而 外部类需销毁时)，将使得外部类无法被垃圾回收器 (GC) 回收，从而造成 内存泄露 | • Handler类 = 静态内部类 + 弱引用持有Activity实例 • 当外部类结束生命周期时，清空Handler内消息队列 | / |
| 资源对象使用后未关闭 | | 对于资源的使用，若在Activity销毁时无及时关闭 / 注销这些资源，则这些资源将不会被回收，从而造成内存泄漏 (资源 如 广播BroadcastReceiver、文件流File、数据库游标Cursor、图片资源Bitmap等) | 在Activity销毁时 及时关闭 / 注销资源 | / |
| 其他使用 | Context | 当 拥有该Activity Context参数对象仍在使用 、而 该Activity需销毁时，该Activity则由于被保持引用而 无法被回收，从而造成内存泄露 (当拥有Context参数的对象的生命周期 > 该Context参数的生命周期时，则容易出现内存泄露) | 对Context的引用 不要超过它本身的生命周期 • 如：尽量使用ApplicationContext代替ActivityContext • 因ApplicationContext会随着应用程序的存在而存在，而不依赖于activity的生命周期 | • 在Android中，通常可使用Context对象有2种：Activity、Application • 当类 / 方法需Context对象时，常优先使用 Activity作为Context参数；此时该对象对整个Activity保持引用 |
| | WebView | 不再使用WebView对象后 无销毁，导致占用的内存长期无法被回收，从而造成内存泄露 | 通过多线程在不使用WebView对象时进行销毁 | 如： 1. 为WebView开启另一个进程 2. 通过IDL与主线程进行通信，WebView所在的进程可根据业务的需要，选择在合适的时机销毁，从而达到内存的完整释放 |
| | Adapter | 在滑动ListView获取最新的View时，每次都在getView()中重新实例化一个View对象，不仅浪费资源、时间，也将使得内存占用越来越大，从而使得内存泄露 | • 使用缓存的convertView • 直接使用 ViewHolder | 1. 初始化时，ListView会根据当前的屏幕布局，从Adapter中实例化一定数量的View对象，同时ListView会将这些view对象缓存起来 2. 当向上、下滚动ListView时，最先位于最上、下面的list item的View对象会回收，然后被用来构造新出现的最下面的list item。 a. 这个构造过程由getView()方法完成：向ListView提供每一个item所需要的view对象 b. getView()的第二个形参View convertView = 被缓存起来的list item的view对象 c. 初始化时缓存中没有view对象，则convertView = null |

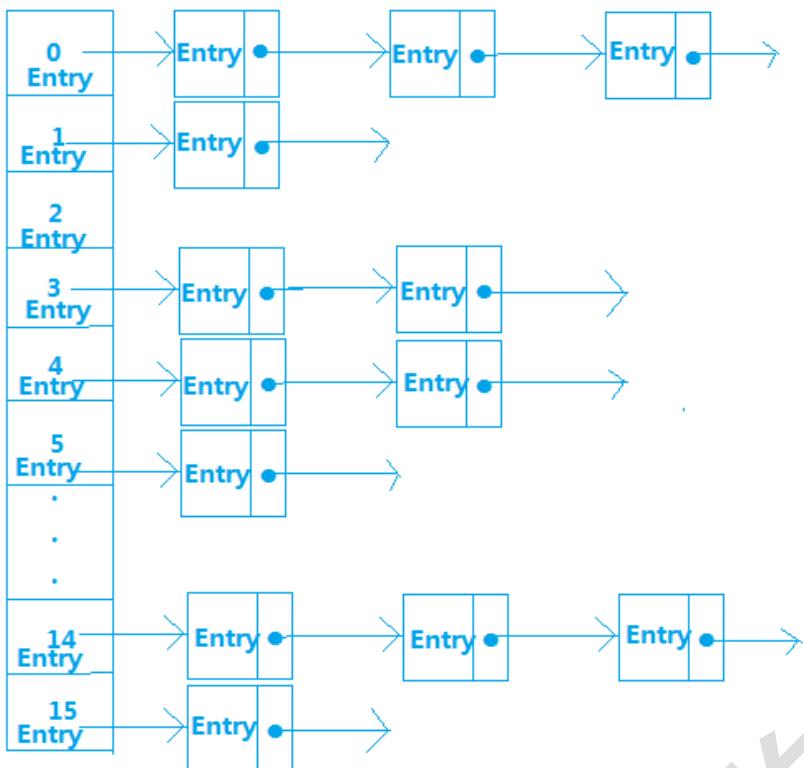
2.6 Android内存优化（使用SparseArray和ArrayMap代替HashMap）

在Android开发时，我们使用的大部分都是Java的api，比如HashMap这个api，使用率非常高，但是对于Android这种对内存非常敏感的移动平台，很多时候使用一些java的api并不能达到更好的性能，相反反而更消耗内存，所以针对Android这种移动平台，也推出了更符合自己的api，比如SparseArray、ArrayMap用来代替HashMap在有些情况下能带来更好的性能提升。

介绍它们之前先来介绍一下HashMap的内部存储结构，就明白为什么推荐使用SparseArray和ArrayMap

HashMap

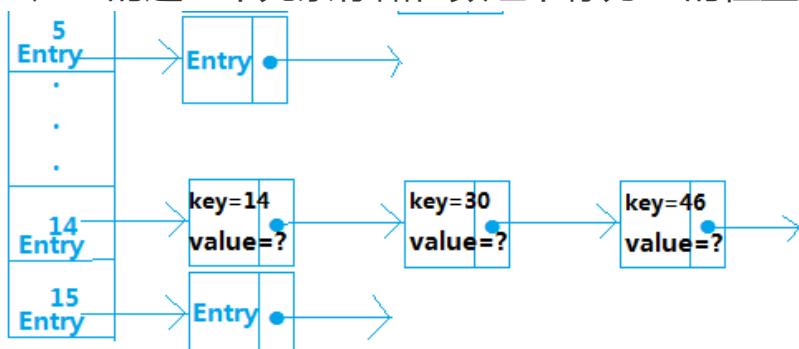
HashMap内部是使用一个默认容量为16的数组来存储数据的，而数组中每一个元素却又是一个链表的头结点，所以，更准确的来说，HashMap内部存储结构是使用哈希表的拉链结构（数组+链表），如图：这种存储数据的方法叫做拉链法



且每一个结点都是Entry类型，那么Entry是什么呢？我们来看看HashMap中Entry的属性：

```
final K key;
V value;
final int hash;
HashMapEntry<K, V> next;
```

从中我们得知Entry存储的内容有key、value、hash值、和next下一个Entry，那么，这些Entry数据是按什么规则进行存储的呢？就是通过计算元素key的hash值，然后对HashMap中数组长度取余得到该元素存储的位置，计算公式为 $\text{hash}(\text{key}) \% \text{len}$ ，比如：假设 $\text{hash}(14)=14, \text{hash}(30)=30, \text{hash}(46)=46$ ，我们分别对len取余，得到 $\text{hash}(14)\%16=14, \text{hash}(30)\%16=14, \text{hash}(46)\%16=14$ ，所以key为14、30、46的这三个元素存储在数组下标为14的位置，如：



从中可以看出，如果有多个元素key的hash值相同的话，后一个元素并不

会覆盖上一个元素，而是采取链表的方式，把之后加进来的元素加入链表末尾，从而解决了hash冲突的问题，由此我们知道HashMap中处理hash冲突的方法是链地址法，在此补充一个知识点，处理hash冲突的方法有以下几种：

1. 开放地址法
2. 再哈希法
3. 链地址法
4. 建立公共溢出区

讲到这里，重点来了，我们知道HashMap中默认的存储大小就是一个容量为16的数组，所以当我们创建出一个HashMap对象时，即使里面没有任何元素，也要分别一块内存空间给它，而且，我们再不断的向HashMap里put数据时，当达到一定的容量限制时（这个容量满足这样的一个关系时候将会扩容：HashMap中的数据量>容量*加载因子，而HashMap中默认的加载因子是0.75），HashMap的空间将会扩大，而且扩大后新的空间一定是原来的2倍，我们可以看put()方法中有这样的一行代码：

```
int newCapacity = oldCapacity * 2;
```

所以，重点就是这个，只要一满足扩容条件，HashMap的空间将会以2倍的规律进行增大。假如我们有几十万、几百万条数据，那么HashMap要存储完这些数据将要不断的扩容，而且在此过程中也需要不断的做hash运算，这将对我们的内存空间造成很大消耗和浪费，而且HashMap获取数据是通过遍历Entry[]数组来得到对应的元素，在数据量很大时候会比较慢，所以在Android中，HashMap是比较费内存的，我们在一些情况下可以使用SparseArray和ArrayMap来代替HashMap。

SparseArray

SparseArray比HashMap更省内存，在某些条件下性能更好，主要是因为它避免了对key的自动装箱（int转为Integer类型），它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间，我们从源码中可以看到key和value分别是用数组表示：

```
private int[] mKeys;
private Object[] mValues;
```

我们可以看到，SparseArray只能存储key为int类型的数据，同时，SparseArray在存储和读取数据时候，使用的是二分查找法，我们可以看看：

```
public void put(int key, E value) {
    int i = ContainerHelpers.binarySearch(mKeys,
mSize, key);
    ...
}
public E get(int key, E valueIfKeyNotFound) {
    int i = ContainerHelpers.binarySearch(mKeys,
mSize, key);
    ...
}
```

也就是在put添加数据的时候，会使用二分查找法和之前的key比较当前我们添加的元素的key的大小，然后按照从小到大的顺序排列好，所以，SparseArray存储的元素都是按元素的key值从小到大排列好的。而在获取数据的时候，也是使用二分查找法判断元素的位置，所以，在获取数据的时候非常快，比HashMap快的多，因为HashMap获取数据是通过遍历Entry[]数组来得到对应的元素。

添加数据

```
public void put(int key, E value)
```

删除数据

```
public void remove(int key)
```

or

```
public void delete(int key)
```

其实remove内部还是通过调用delete来删除数据的

获取数据

```
public E get(int key)
```

or

```
public E get(int key, E valueIfKeyNotFound)
```

该方法可设置如果key不存在的情况下默认返回的value

特有方法

在此之外，SparseArray还提供了两个特有方法，更方便数据的查询：

获取对应的key：

```
public int keyAt(int index)
```

获取对应的value：

```
public E valueAt(int index)
```

SparseArray应用场景：

虽说SparseArray性能比较好，但是由于其添加、查找、删除数据都需要先进行一次二分查找，所以在数据量大的情况下性能并不明显，将降低至少50%。

满足下面两个条件我们可以使用SparseArray代替HashMap：

- 数据量不大，最好在千级以内
- key必须为int类型，这中情况下的HashMap可以用SparseArray代替：

```
HashMap<Integer, Object> map = new HashMap<>();
```

用SparseArray代替：

```
SparseArray<Object> array = new SparseArray<>();
```

ArrayMap

这个api的资料在网上可以说几乎没有，然并卵，只能看文档了
ArrayMap是一个`<key,value>`映射的数据结构，它设计上更多的是考虑内存的优化，内部是使用两个数组进行数据存储，一个数组记录key的hash值，另外一个数组记录Value值，它和SparseArray一样，也会对key使用二分法进行从小到大排序，在添加、删除、查找数据的时候都是先使用二分查找法得到相应的index，然后通过index来进行添加、查找、删除等操作，所以，应用场景和SparseArray的一样，如果在数据量比较大的情况下，那么它的性能将退化至少50%。

添加数据

```
public V put(K key, V value)
```

获取数据

```
public V get(Object key)
```

删除数据

```
public V remove(Object key)
```

特有方法

它和SparseArray一样同样也有两个更方便的获取数据方法：

```
public K keyAt(int index)  
public V valueAt(int index)
```

ArrayMap应用场景

- 数据量不大，最好在千级以内
- 数据结构类型为Map类型

```
ArrayMap<Key, Value> arrayMap = new ArrayMap<>();
```

【注】：如果我们要兼容api19以下版本的话，那么导入的包需要为v4包

```
import android.support.v4.util.ArrayMap;
```

总结

SparseArray和ArrayMap都差不多，使用哪个呢？

假设数据量都在千级以内的情况下：

- 1、如果key的类型已经确定为int类型，那么使用SparseArray，因为它避免了自动装箱的过程，如果key为long类型，它还提供了一个LongSparseArray来确保key为long类型时的使用
- 2、如果key类型为其它的类型，则使用ArrayMap

第三节 绘制优化

什么情况下使用 ViewStub、include、merge？

他们的原理是什么？

3.1 Android性能优化：那些不可忽略的绘制优化

前言

- 在Android开发中，性能优化策略十分重要
- 本文主要讲解性能优化中的**绘制优化**，希望你们会喜欢。
- ○ *

3.1.1 影响的性能

绘制性能的好坏 主要影响：**Android应用中的页面显示速度**

3.1.2 如何影响性能

绘制影响**Android**性能的实质：**页面的绘制时间**

1个页面通过递归 完成测量 & 绘制过程

3.1.3 优化思路

主要优化方向是：

1. 降低 `view.onDraw()` 的复杂度
2. 避免过度绘制 (Overdraw)

3.1.4 具体优化方案

- 具体如下

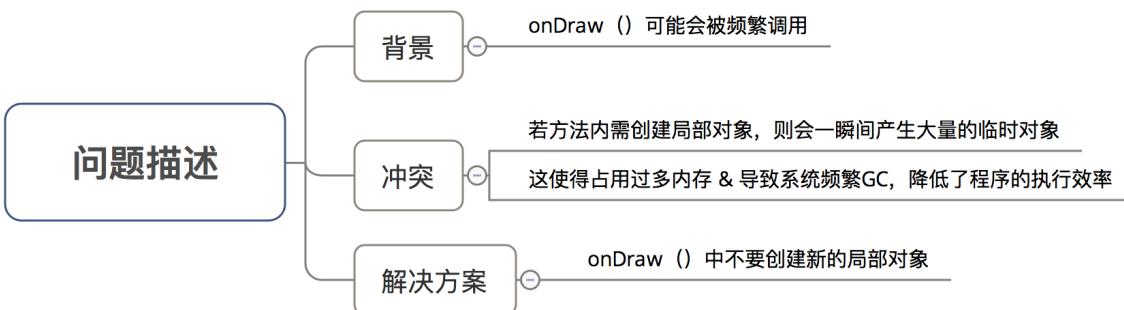


示意图

- 下面，我将详细分析每种优化方案

4.1. 降低 `View.onDraw()` 的复杂度

4.1.1 `onDraw()` 中不要创建新的局部对象



示意图

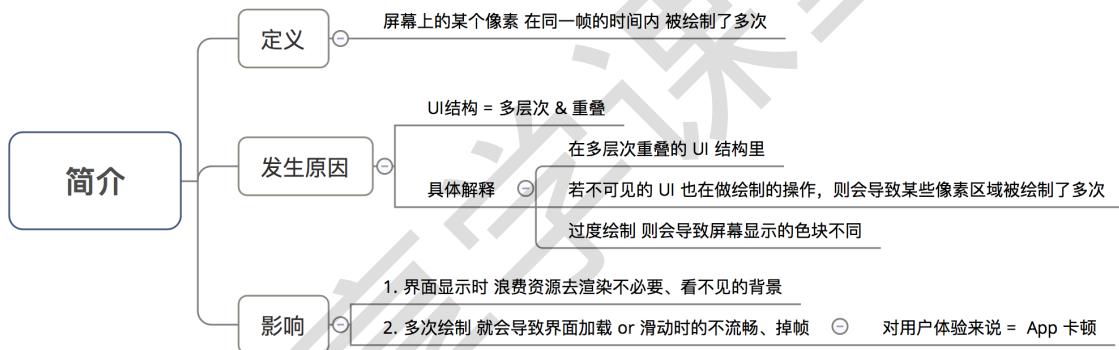
4.1.2 避免onDraw () 执行大量 & 耗时操作



示意图

4.2 避免过度绘制 (Overdraw)

4.2.1 过度绘制的简介



示意图

4.2.2 过度绘制的表现形式

过度绘制 会导致屏幕显示的色块不同, 具体如下



示意图

附：示例说明

如右图

“调试 GPU 过度绘制”的文本显示为蓝色，则代表1次过度绘制

说明

- 背景 = 白色 = 绘制1次
- 文字 = 黑色 = 绘制1次
- 从而导致文字所在区域绘制了2次，即 1次过度绘制



示意图

4.2.3 过度绘制的优化原则

很多过度绘制是难以避免的，如上述实例的文字 & 背景导致的过度绘制，只能尽可能避免过度绘制：

1. 尽可能地控制过度绘制的次数 = 2 次（绿色）以下，蓝色最理想
2. 尽可能避免过度绘制的粉色 & 红色情况
3. 不允许 3 次以上的过度绘制（淡红色）面积超过屏幕大小的 1/4

4.2.4 优化方案

1. 移除默认的 window 背景
2. 移除控件中不必要的背景
3. 减少布局文件的层级（嵌套）
4. 自定义控件 View 优化：使用 clipRect()、quickReject()

优化方案1：移除默认的 Window 背景

- 背景：一般应用程序默认继承的主题 = `windowBackground`，如默认的 Light 主题：

```
<style name="Theme.Light">
    <item name="isLightTheme">true</item>
    <item
        name="windowBackground">@drawable/screen_background_selector_light</item>
    ...
</style>
```

- 问题：一般情况下，该默认的 Window 背景基本用不上：因背景都自定义设置
若不移除，则导致所有界面都多 1 次绘制
- 解决方案：移除默认的 Window 背景

```
// 方式1：在应用的主题中添加如下的一行属性
<item
name="android:windowBackground">@android:color/transparent</item>
<!-- 或者 -->
<item name="android:windowBackground">@null</item>

// 方式2：在 BaseActivity 的 onCreate() 方法中使用下面的代码移除
getwindow().setBackgroundDrawable(null);
<!-- 或者 -->

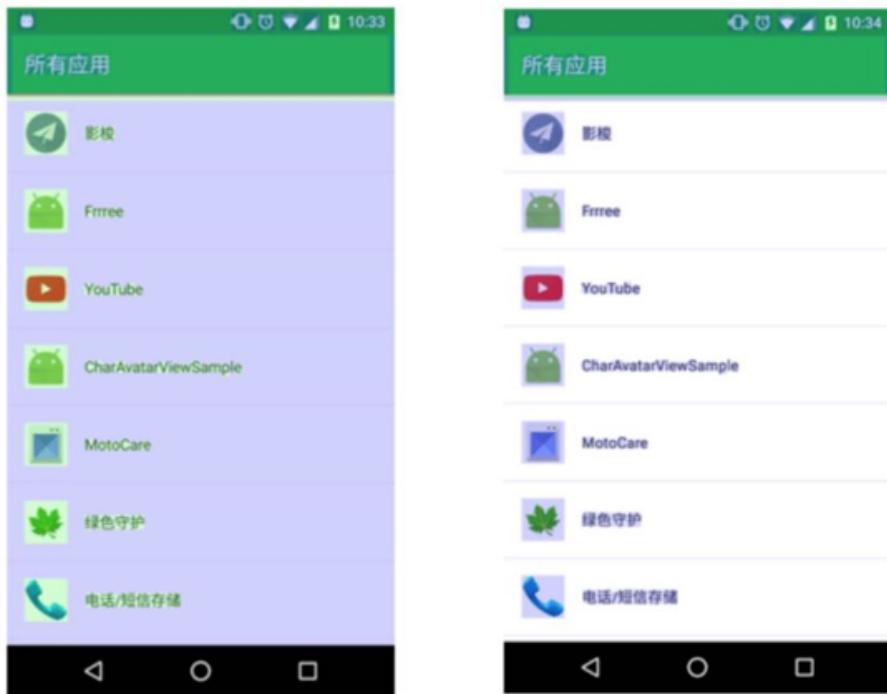
getwindow().setBackgroundDrawableResource(android.R.color
.transparent);
```

优化方案2：移除控件中不必要的背景

如2个常见场景：

- 场景1：ListView 与 Item

列表页（ListView）与其内子控件（Item）的背景相同 = 白色，故可移除子控件（Item）布局中的背景



优化前

- 子控件 = 蓝色 = 1次过度绘制 = 自身背景绘制 + 列表背景绘制
- 列表 = 原色 = 无过度绘制 = 自身列表背景

优化后

- 子控件 = 蓝色 = 无过度绘制 = 列表背景绘制
- 列表 = 原色 = 无过度绘制 = 自身列表背景

示意图

- 场景2: ViewPager 与 Fragment

对于1个ViewPager + 多个Fragment组成的首页界面，若每个Fragment都设有背景色，即ViewPager则无必要设置，可移除



示意图

关于更多场景，可使用工具 `Hierarchy View` 查看，具体请看文章：[过渡绘制的使用工具：Hierarchy View](#)

优化方案3：减少布局文件的层级（减少不必要的嵌套）

- 原理：减少不必要的嵌套 -> UI层级少 -> 过度绘制的可能性低
- 优化方式：使用布局标签 `<merge>` & 合适选择布局类型

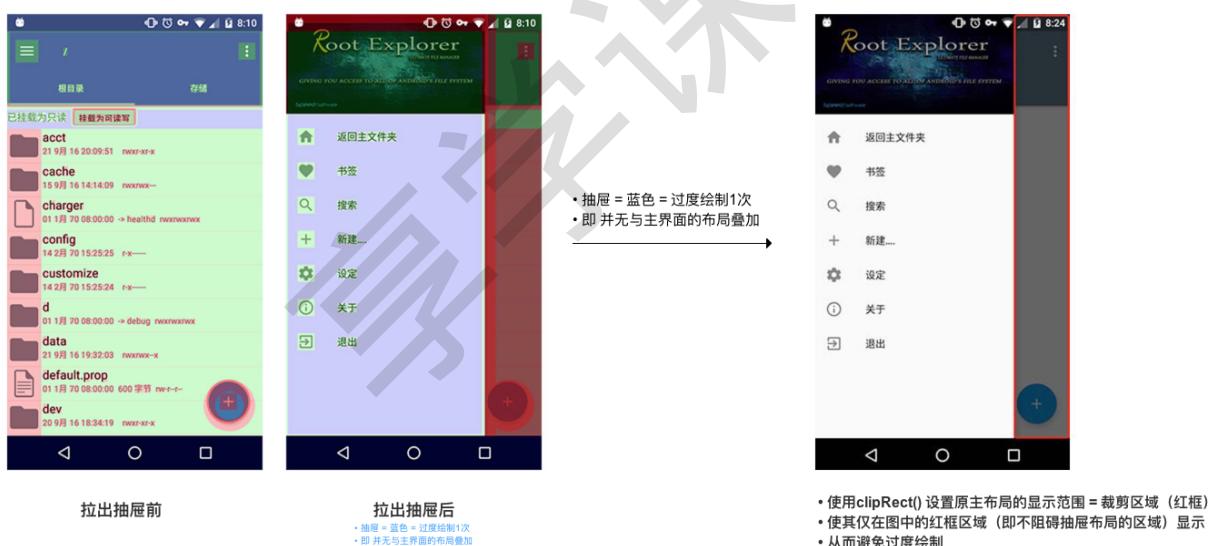
具体请看文章：[Android性能优化：这是一份详细的布局优化指南（含、）](#)

优化方案4：自定义控件View优化：使用 `clipRect()`、`quickReject()`

- `clipRect()`

1. 作用：给 `Canvas` 设置一个裁剪区域，只有在该区域内才会被绘制，区域之外的都不绘制
2. 实例说明：`DrawerLayout` 布局 = 左抽屉布局

`DrawerLayout` 布局的过度绘制扫描结果



示意图

```
@Override  
protected boolean drawChild(Canvas canvas, View child,  
long drawingTim  
// ...仅贴出关键代码
```

```
// 1. 遍历 DrawerLayout 的 child view, 拿到抽屉布局  
for (int i = 0; i < childCount; i++) {  
    final View v = getChildAt(i);
```

```

        if (v == child || v.getVisibility() != VISIBLE
            || !hasOpaqueBackground(v) ||
            !isDrawerView(v)
            || v.getHeight() < height) {
            continue;
        }

        // a. 若是左抽屉布局
        // 则取抽屉布局的右边界作为裁剪区的左边界、设置原主布
        局的裁剪区域，如上图裁剪区域
        if (checkDrawerViewAbsoluteGravity(v,
            Gravity.LEFT)) {
            final int vright = v.getRight();
            if (vright > clipLeft) clipLeft = vright;
        }
        // b. 若是右抽屉布局
        // 则取抽屉布局的左边界作为裁剪区的右边界、设置原主布
        局的裁剪区域
    } else {
        final int vleft = v.getLeft();
        if (vleft < clipRight) clipRight = vleft;
    }
}

// 2. 通过clipRect() 设置原主布局的显示范围 = 裁剪区
域，使其仅在上图中的红框区域（即不阻碍抽屉布局的区域）显示
// 从而避免过度绘制
canvas.clipRect(clipLeft, 0, clipRight,
getHeight());
}
}

```

- `quickreject()`

1. 作用：判断和某个矩形相交
2. 具体措施：若判断与矩形相交，则可跳过相交的区域，从而减少过
度绘制

4.3 其他优化方案

| 优化方案 | 具体描述 | 备注 |
|---------------------------------|---|-----------------|
| 使用OpenGL绘图 (Android最高级的绘图机制) | Android支持使用 OpenGL API的高性能绘图 | 支持OpenGL ES 3.0 |
| 尽量为所有分辨率创建资源 | 资源匹配分辨率 = 减少不必要的缩放，从而提高UI绘制效率 | |
| 使用语言RenderJavascript | <ul style="list-style-type: none"> 在Android上写高性能代码的一种语言 语法给予C语言的C99标准 其结构独立的，故不需为不同的CPU或者GPU定制代码代码 | Adnroid3.0引进 |

示意图

总结

| 优化方向 | 优化原因 | 优化方案 |
|-------------------|--|---|
| 降低 onDraw () 的复杂度 | <ul style="list-style-type: none"> onDraw () 可能会被频繁调用 若方法内需创建局部对象，则会一瞬间产生大量的临时对象 这使得占用过多内存 & 导致系统频繁GC，降低了程序的执行效率 | onDraw () 中不要创建新的局部对象 |
| | <ul style="list-style-type: none"> Google官方性能优化标准：View的最佳绘制频率 = 60fps View.onDraw () 执行大量 & 耗时操作，会抢占CPU的时间片 从而导致 View的绘制过程不流畅 | 避免onDraw () 执行大量 & 耗时操作 (放到其他函数中执行、多线程) |
| 避免过度绘制 | <ul style="list-style-type: none"> 一般情况下，该默认的Window背景基本用不上(因背景都自定义设置) 若不移除，则导致所有界面都多1次绘制 | 移除默认的 Window 背景 |
| | 控件与控件 or 控件与布局之间的背景重复，从而导致二次绘制 | 移除 控件中不必要的背景 |
| | 布局文件的层级过多会导致过度绘制 | 使用布局标签<merge> & 合适选择布局类型 减少不必要的嵌套 |
| 其他 | 使用OpenGL绘图 (Android最高级的绘图机制) | Android支持使用 OpenGL API的高性能绘图 (支持OpenGL ES 3.0) |
| | 尽量为所有分辨率创建资源 | 资源匹配分辨率 = 减少不必要的缩放，从而提高UI绘制效率 |
| | 使用语言RenderJavascript (Adnroid3.0引进) | <ul style="list-style-type: none"> 在Android上写高性能代码的一种语言 语法给予C语言的C99标准 其结构独立的，故不需为不同的CPU或者GPU定制代码代码 |

示意图

至此，关于绘制优化的方案讲解完毕。

3.1.5 布局调优工具

- 背景

尽管已经注意到上述的优化策略，但实际开发中难免还是会出现布局

性能的问题

- 解决方案

使用 布局调优工具

此处主要介绍 常用的： `hierarchy viewer`、 `Profile GPU Rendering`、 `Systrace`

5.1 Hierarchy Viewer

- 简介

`Android Studio` 提供的UI性能检测工具。

- 作用

可视化获得UI布局设计结构 & 各种属性信息，帮助我们优化布局设计

即：方便查看 `Activity` 布局，各个 `view` 的属性、布局测量-布局-绘制的时间

- 具体使用

[Hierarchy Viewer 使用指南](#)

5.2 Profile GPU Rendering

- 简介

一个 图形监测工具

- 作用

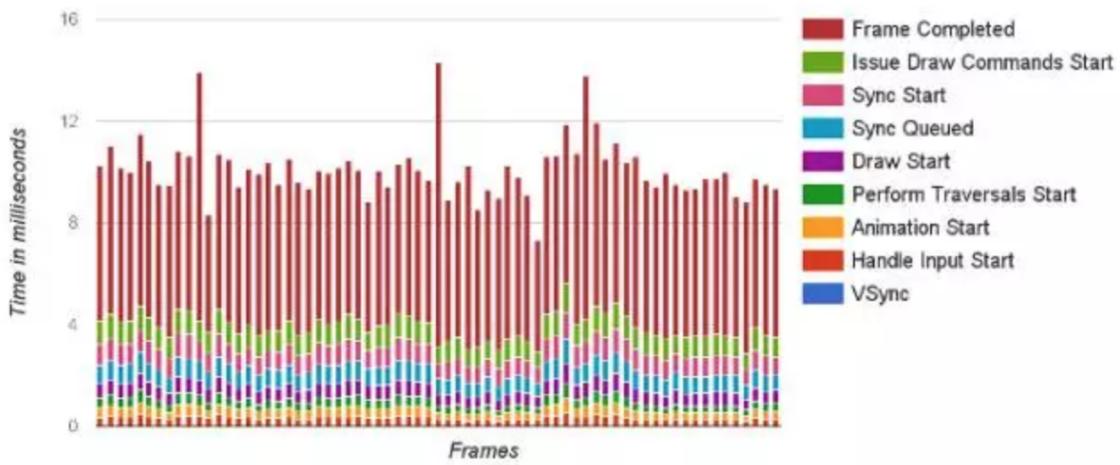
渲染、绘制性能追踪

能实时反应当前绘制的耗时

- 具体使用

横轴 = 时间、纵轴 = 每帧的耗时；随着时间推移，从左到右的刷新呈现

提供一个标准的耗时，如果高于标准耗时，就表示当前这一帧丢失



示意图

更详细使用请看： [Profile GPU Rendering 使用指南](#)

5.3 Systrace

- 简介

Android 4.1以上版本提供的性能数据采样 & 分析工具

- 作用

检测 Android 系统各个组件随着时间的运行状态 & 提供解决方案

1. 收集 等运行信息，从而帮助开发者更直观地分析系统瓶颈，改进性能

检测范围包括：Android 关键子系统（如 `windowManagerService` 等 `Framework` 部分关键模块）、服务、View 系统

2. 功能包括：跟踪系统的 I/O 操作、内核工作队列、CPU 负载等，在 UI 显示性能分析上提供很好的数据，特别是在动画播放不流畅、渲染卡等问题上

- 具体使用

[Systrace 使用指南](#)

- ○ *

3.1.6 总结

- 本文主要讲解 Android 性能优化中的 绘制优化

| 优化方向 | 优化原因 | 优化方案 |
|-------------------|--|---|
| 降低 onDraw () 的复杂度 | <ul style="list-style-type: none"> onDraw () 可能会被频繁调用 若方法内需创建局部对象，则会一瞬间产生大量的临时对象 这使得占用过多内存 & 导致系统频繁GC，降低了程序的执行效率 | onDraw () 中不要创建新的局部对象 |
| | <ul style="list-style-type: none"> Google官方性能优化标准：View的最佳绘制频率 = 60fps View.onDraw () 执行大量 & 耗时操作，会抢占CPU的时间片 从而导致 View的绘制过程不流畅 | 避免onDraw () 执行大量 & 耗时操作 (放到其他函数中执行、多线程) |
| 避免过度绘制 | <ul style="list-style-type: none"> 一般情况下,该默认的Window背景基本用不上(因背景都自定义设置) 若不移除，则导致所有界面都多 1 次绘制 | 移除默认的 Window 背景 |
| | 控件与控件 or 控件与布局之间的背景重复，从而导致二次绘制 | 移除 控件中不必要的背景 |
| | 布局文件的层级过多会导致过度绘制 | 使用布局标签<merge> & 合适选择布局类型 减少不必要的嵌套 |
| 其他 | 使用OpenGL绘图 (Android最高级的绘图机制) | 提高绘制性能 Android支持使用 OpenGL API的高性能绘图 (支持OpenGL ES 3.0) |
| | 尽量为所有分辨率创建资源 | 资源匹配分辨率 = 减少不必要的缩放，从而提高UI绘制效率 |
| | 使用语言RenderJavascript (Android3.0引进) | <ul style="list-style-type: none"> 在Android上写高性能代码的一种语言 语法给予C语言的C99标准 其结构独立的，故不需为不同的CPU或者GPU定制代码代码 |

示意图

第四节 安装包优化

4.1 Android 安装包优化--减小安装包体积

Android的安装包APK文件本身就是个压缩文件。把后缀名改成.zip，用解压软件解压后，就能看到安装包的内容。



Jietu20190303-124224.jpg

从上图可以看到，安装包内的文件结构是：

1. assets 资源文件
2. res 资源文件
3. *.dex Dalvik字节码
4. META-INF 签名信息
5. lib so库

要减小安装包的体积本质就是要减少包内的文件的大小。接下来我们依次来看如何减少这些文件的大小。

图片资源

1. 能用shape就绝不用图片。对于纯色或渐变的图片，能用shape渲染的就优先使用shape。不仅文件体积小，还渲染速度快，也不用考虑适配问题。
2. 图片压缩。一般UI给的资源图都是软件直接导出的PNG图片，体积较大。可以把图片压缩后，再放入项目中。推荐使用[tinyPNG](#)，或者AndroidStudio 插件[TinyPic](#) 压缩。
3. 使用webp图片。相同的图片质量下，体积更小。
 - webp的无损压缩比PNG文件小45%左右
 - AndroidStudio 自带转换功能，可以把png转换为webp。
 - 不用考虑Android系统兼容问题，因为目前大部分App的最低支持版本都是4.4.4。
 - 缺点是解码速度相对慢些，在加载网络图片上很有优势。
4. svg矢量图。其实是图片的描述文件，牺牲CPU的计算能力的，节省空间。适用于简单的图标。

目前WEBP与JPG相比较，编码速度慢10倍，解码速度慢1.5倍，虽然会增加额外的解码时间，但是由于减少了文件体积，缩短了加载的时间，实际上文件的渲染速度反而变快了。

4.2.1+ 对于webp的decode、encode是完全支持的（包含半透明的webp图）

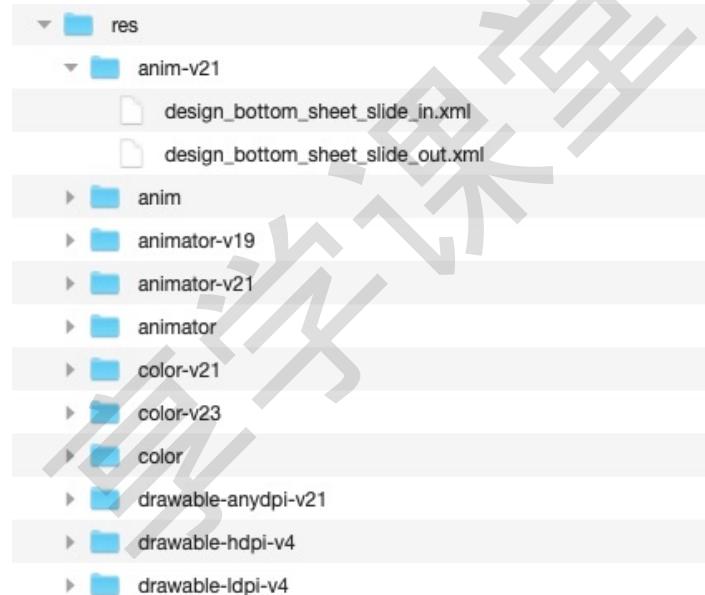
对于4.0+ 到 4.2.1，只支持完全不透明的decode、encode的webp图
4.0 以下，应该是默认不支持webp了

资源文件混淆

主要是使用微信开源的[AndResGuard](#)资源混淆工具。这是个缩小APK大小的工具，他的原理类似Java Proguard，但是只针对资源。他会将原本冗长的资源路径变短，例如将res/drawable/wechat变为r/d/a。

这个工具不仅可以混淆资源文件，还利用了7z深度压缩，进一步减少安装包大小。如何使用请参考 [AndResGuard的使用](#)。

实现的原理是：resource.arsc文件中保存着资源文件夹中各个资源的路径。AndResGuard工具将资源文件重命名(甚至修改文件夹名，将drawable改为a0)后，再生成新的resource.arsc文件，替换源文件打包签名即可。



混淆之前的资源文件夹



混淆之后的资源文件夹

去除不用so文件

在使用一些三方库的时候，会集成大量的so文件到项目中，这些so文件都对应着不同的CPU架构。

Android系统目前支持以下七种不同的CPU架构：ARMv5、ARMv7、x86、MIPS、ARMv8、MIPS64、x86_64，每一个CPU架构对应一个ABI：armeabi, armeabi-v7a, x86, mips, arm64-v8a, mips64, x86_64。

目前市面上绝大部分的CPU架构都是 ARMv7/ARMv8，所以可以在gradle中加入配置，只保留v7,v8。

```
defaultConfig {  
    ...  
    ndk {  
        abiFilters 'armeabi-v7a', 'arm64-v8a'  
    }  
}
```

甚至只保留 armeabi-v7a，微信大佬就这么干的！

资源动态加载

也可以将一些不必要资源放入网络中，等需要时再下载，比如：

- emoji表情
- 换肤
- 动态添加插件化模块([插件化框架](#))

移除无用资源

可以使用Lint工具检测无用的资源，然后移除掉。

1. 点击菜单“Analyze”
2. 选择 “run inspection by Name ...”
3. 在弹出的搜索窗口中输入想执行的检查类型，如“Unused Resources”。

4. 选择检查的范围，一般选择整个项目或模块。
5. 选择好后点ok就开始检查了。然后就可以根据检查结果来去除重复的资源。

如果觉得一个个删除太麻烦，可以开启代码混淆，在打包时排除不需要的资源。

代码混淆

这个不用多说，大家都知道。

1. 可以删除注释和不用的代码。
2. 将java文件名改成短名a.java, b.java

```
buildTypes {  
    release {  
        buildConfigField "boolean", "LOG_DEBUG",  
        "false" //不显示log  
        minifyEnabled true //开启混淆  
        shrinkResources true //移除无用资源  
        zipAlignEnabled true //zipalign优化  
        proguardFiles  
            getDefaultProguardFile('proguard-android.txt'),  
            'proguard-rules.pro'  
            signingConfig signingConfigs.config  
    }  
}  
}
```

Android的安装包APK文件本身就是个压缩文件。把后缀名改成.zip，用解压软件解压后，就能看到安装包的内容。



从上图可以看到，安装包内的文件结构是：

1. assets 资源文件
2. res 资源文件
3. *.dex Dalvik字节码
4. META-INF 签名信息
5. lib so库

要减小安装包的体积本质就是要减少包内的文件的大小。接下来我们依次来看如何减少这些文件的大小。

图片资源

1. 能用shape就绝不用图片。对于纯色或渐变的图片，能用shape渲染的就优先使用shape。不仅文件体积小，还渲染速度快，也不用考虑适配问题。
2. 图片压缩。一般UI给的资源图都是软件直接导出的PNG图片，体积较大。可以把图片压缩后，再放入项目中。推荐使用[tinyPNG](#)，或者AndroidStudio 插件[TinyPic](#) 压缩。
3. 使用webp图片。相同的图片质量下，体积更小。
 - webp的无损压缩比PNG文件小45%左右
 - AndroidStudio 自带转换功能，可以把png转换为webp。
 - 不用考虑Android系统兼容问题，因为目前大部分App的最低支持版本都是4.4.4。
 - 缺点是解码速度相对慢些，在加载网络图片上很有优势。
4. svg矢量图。其实是图片的描述文件，牺牲CPU的计算能力的，节省空间。适用于简单的图标。

目前WEBP与JPG相比较，编码速度慢10倍，解码速度慢1.5倍，虽然会增加额外的解码时间，但是由于减少了文件体积，缩短了加载的时间，实际上文件的渲染速度反而变快了。

4.2.1+ 对于webp的decode、encode是完全支持的（包含半透明的webp图）

对于4.0+ 到 4.2.1，只支持完全不透明的decode、encode的webp图

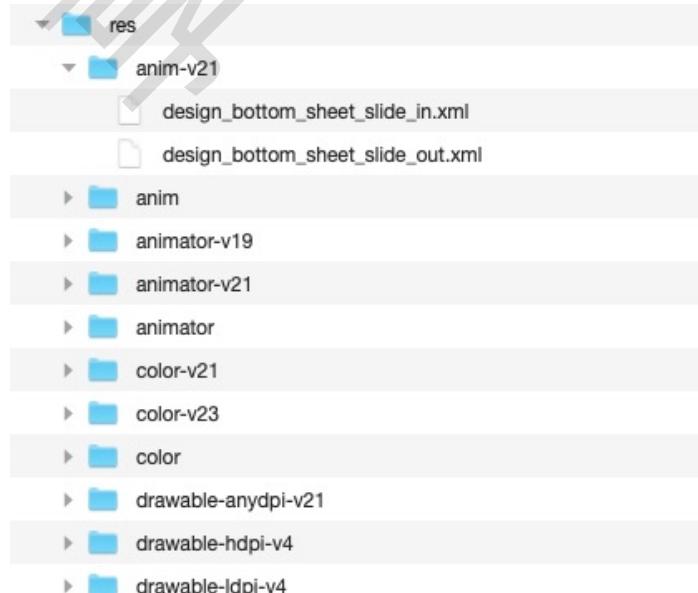
4.0 以下，应该是默认不支持webp了

资源文件混淆

主要是使用微信开源的[AndResGuard](#)资源混淆工具。这是个缩小APK大小的工具，他的原理类似Java Proguard，但是只针对资源。他会将原本冗长的资源路径变短，例如将res/drawable/wechat变为r/d/a。

这个工具不仅可以混淆资源文件，还利用了7z深度压缩，进一步减少安装包大小。如何使用请参考 [AndResGuard的使用](#)。

实现的原理是：resource.arsc文件中保存着资源文件夹中各个资源的路径。AndResGuard工具将资源文件重命名(甚至修改文件夹名，将drawable改为a0)后，再生成新的resource.arsc文件，替换源文件打包签名即可。



混淆之前的资源文件夹



混淆之后的资源文件夹

去除不用so文件

在使用一些三方库的时候，会集成大量的so文件到项目中，这些so文件都对应着不同的CPU架构。

Android系统目前支持以下七种不同的CPU架构：ARMv5、ARMv7、x86、MIPS、ARMv8、MIPS64、x86_64，每一个CPU架构对应一个ABI：armeabi, armeabi-v7a, x86, mips, arm64-v8a, mips64, x86_64。

目前市面上绝大部分的CPU架构都是 ARMv7/ARMv8，所以可以在gradle中加入配置，只保留v7,v8。

```
defaultConfig {  
    ...  
    ndk {  
        abiFilters 'armeabi-v7a', 'arm64-v8a'  
    }  
}
```

甚至只保留 armeabi-v7a，微信大佬就这么干的！

资源动态加载

也可以将一些不必要资源放入网络中，等需要时再下载，比如：

- emoji表情

- 换肤
- 动态添加插件化模块([插件化框架](#))

移除无用资源

可以使用Lint工具检测无用的资源，然后移除掉。

1. 点击菜单“Analyze”
2. 选择 “run inspection by Name ...”
3. 在弹出的搜索窗口中输入想执行的检查类型，如“Unused Resources”。
4. 选择检查的范围，一般选择整个项目或模块。
5. 选择好后点ok就开始检查了。然后就可以根据检查结果来去除重复的资源。

如果觉得一个个删除太麻烦，可以开启代码混淆，在打包时排除不需要的资源。

代码混淆

这个不用多说，大家都知道。

1. 可以删除注释和不用的代码。
2. 将java文件名改成短名a.java, b.java

```
buildTypes {  
    release {  
        buildConfigField "boolean", "LOG_DEBUG",  
        "false" //不显示log  
        minifyEnabled true //开启混淆  
        shrinkResources true //移除无用资源  
        zipAlignEnabled true //zipalign优化  
        proguardFiles  
            getDefaultProguardFile('proguard-android.txt'),  
            'proguard-rules.pro'  
            signingConfig signingConfigs.config  
    }  
}  
}
```

4.2 稳定性优化

第七章 源码流程篇

第一节 开源库源码分析

RxJava的实现原理，它是如何实现线程的控制？

Retrofit的框架结构是什么？底层是怎么实现的？

网络框架是如何搭建？okhttp的底层实现是什么，和Retrofit有什么不同？

图片加载框架gilde、Picasso、fresco有什么不同，各自的实现原理是什么？如何搭建一个网络框架？

第二节 Glide源码分析

【面试题】

Glide的优点有哪些？

Glide的缓存原理是什么？

Glide的优点主要包括：

- 多种图片格式的缓存，适用于更多的内容表现形式（如Gif、WebP、缩略图、Video）
- 生命周期集成（根据Activity或者Fragment的生命周期管理图片加载请求）Glide可以感知调用页面的生命周期，这就是优势
- 高效处理Bitmap（bitmap的复用和主动回收，减少系统回收压力）
- 高效的缓存策略，灵活（Picasso只会缓存原始尺寸的图片，Glide缓存的是多种规格），加载速度快且内存开销小（默认Bitmap格式的不同，使得内存开销是Picasso的一半）

第三节 Android面试题：Glide

3.1 Glide源码学习一：Glide框架介绍、with方法详解

Glide的几个基本概念：

Model：表示数据来源，因为Glide加载的图片资源类型有很多，可以是string、resourceID、url、file等。不论是是什么资源类型，Glide都会把它封装成对应的Model模型。

Data：数据。从数据源中获取了Model之后，需要把它加工成数据，一般来讲都是inputstream。

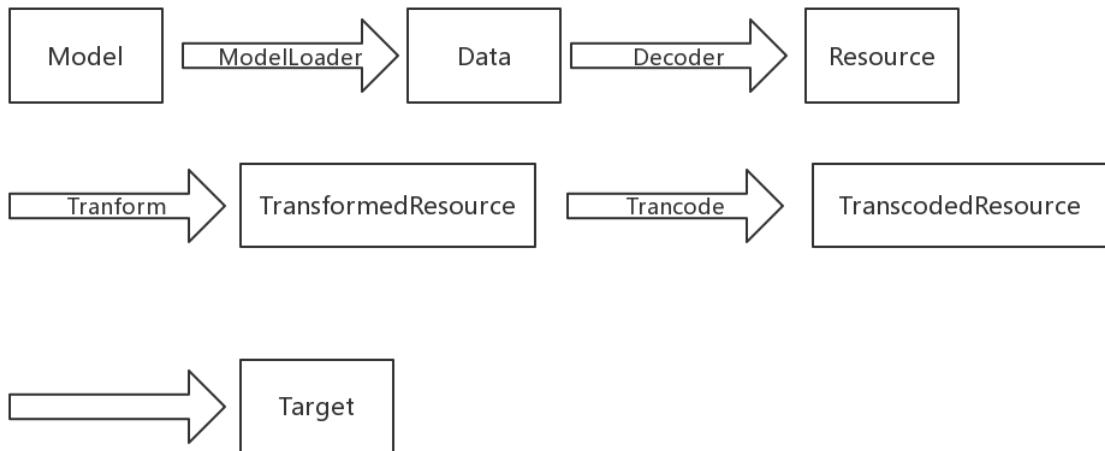
Resource：把inputstream解码成bitmap后，解码之后的Data我们称之为Resource。

TransformedResource（转换）：Resource需要进行裁剪等操作，然后变为TransformedResource。

TranscodedResource（转码）：Glide除了加载静态图片外，还能加载动态图片GIF，这里就需要TranscodedResource。

Target: Glide把我们需要展示的东西封装成target然后放到对应的view上。

我们可以看一个图片:



使用:

```
Glide.with(getApplicationContext()) // 指定Context
    .load(url)// 指定图片的URL
    .placeholder(R.mipmap.ic_launcher)// 指定
    图片未成功加载前显示的图片
    .error(R.mipmap.ic_launcher)// 指定图片加载
    失败显示的图片
    .override(300, 300)//指定图片的尺寸
    .fitCenter()//指定图片缩放类型为
    .centerCrop()// 指定图片缩放类型为
    .skipMemoryCache(true)// 跳过内存缓存

    .diskCacheStrategy(DiskCacheStrategy.NONE)//跳过磁盘缓存

    .diskCacheStrategy(DiskCacheStrategy.SOURCE)//仅仅只缓存原来的
    全分辨率的图像

    .diskCacheStrategy(DiskCacheStrategy.RESULT)//仅仅缓存最终的
    图像
```

```
.diskCacheStrategy(DiskCacheStrategy.ALL)//缓存所有版本的图  
像  
    .priority(Priority.HIGH)//指定优先级.Glide  
将会用他们作为一个准则，  
//并尽可能的处理这些请求，  
    // 但是它不能保证所有的图片都会按照所要求的顺序加  
载。优先级排序：  
    //IMMEDIATE > HIGH > NORMAL > LOW  
    .into(imageview);//指定显示图片的Imageview
```

with方法详解：

with传入的context决定了Glide的生命周期，如果传入的是activity，那么当activity被销毁后，Glide就也会被销毁。如果传入的ApplicationContext，那么它的生命周期就跟APP相同。

因此，可以得出，with方法的主要作用就是将Glide与我们的APP或者组件的生命周期绑定到一起。那么为什么要将Glide与生命周期绑定到一起呢，举例：进入到一个activity后，我们需要加载一张图片，当图片还未加载完成时，我们关闭了这个activity，那么此时，Glide就不需要再去加载图片了，需要关闭了，当我们绑定到生命周期上，就可以非常方便的执行这个操作了。

进入Glide源码，我们可以看到五个with的重载方法：

```
public static RequestManager with(Context context) {  
    RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
    return retriever.get(context);  
}  
  
public static RequestManager with(Activity activity)  
{  
    RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
    return retriever.get(activity);  
}
```

```
public static RequestManager with(FragmentActivity activity) {
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(activity);
}

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public static RequestManager
with(android.app.Fragment fragment) {
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(fragment);
}

public static RequestManager with(Fragment fragment)
{
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(fragment);
}
```

我们先拿第一个方法进行分析：

```
public static RequestManager with(Context context) {
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(context);
}
```

在这里我们看到两个陌生的类RequestManager和RequestManagerRetriever，这两个类有什么作用呢？

RequestManager：可以理解为专门用来管理Glide的一系列加载图片的请求的类。

RequestManagerRetriever：可以理解为专门用来生产、管理和回收RequestManager的类。

```
RequestManagerRetriever retriever =  
RequestManagerRetriever.get();
```

只是获取一个RequestManagerRetriever的单例，没什么好说的。主要来看：

```
return retriever.get(context);
```

点进去 **get(Context context)** :

```
public RequestManager get(Context context) {  
    if (context == null) {  
        throw new IllegalArgumentException("You  
cannot start a load on a null Context");  
    } else if (Util.isOnMainThread() && !(context  
instanceof Application)) {  
        if (context instanceof FragmentActivity) {  
            return get((FragmentActivity) context);  
        } else if (context instanceof Activity) {  
            return get((Activity) context);  
        } else if (context instanceof ContextWrapper)  
{  
            return get((ContextWrapper)  
context).getBaseContext();  
        }  
    }  
  
    return getApplicationManager(context);  
}
```

由于我们传递的是applicationcontext，所以会走到：

```
return getApplicationManager(context);
```

点进去：

```
private RequestManager getApplicationManager(Context context) {
    // Either an application context or we're on a background thread.
    if (applicationManager == null) {
        synchronized (this) {
            if (applicationManager == null) {
                new
RequestManager(context.getApplicationContext(),
                    new ApplicationLifecycle(),
new EmptyRequestManagerTreeNode());
            }
        }
    }

    return applicationManager;
}
```

发现只是获取一个单例而已。到此，可以暂时停止了。因为传入 ApplicationContext并不复杂，没有什么需要判断的，只需要传入 application的lifecycle就可以了，如下：

```
new RequestManager(context.getApplicationContext(),
                    new ApplicationLifecycle(),
new EmptyRequestManagerTreeNode());
```

可以看到new ApplicationLifecycle()。这样就将Glide和Application的声明周期绑定到一起了。

接下来我们来看，非context的get方法：

get(Activity activity)

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public RequestManager get(Activity activity) {
    if (util.isOnBackgroundThread() ||
Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        return get(activity.getApplicationContext());
    } else {
        assertNotDestroyed(activity);
        android.app.FragmentManager fm =
activity.getFragmentManager();
        return fragmentGet(activity, fm);
    }
}
```

我们主要来看这儿的逻辑：

```
android.app.FragmentManager fm =
activity.getFragmentManager();
return fragmentGet(activity, fm);
```

点击fragmentGet()方法：

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
RequestManager fragmentGet(Context context,
android.app.FragmentManager fm) {
    RequestManagerFragment current =
getRequestManagerFragment(fm);
    RequestManager requestManager =
current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context,
current.getLifecycle(),
current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}
```

我们进行详细分析，先看第一行：

```
RequestManagerFragment current =  
getRequestManagerFragment(fm);
```

在这里我们创建了一个RequestManagerFragment对象，很明显这是一个Fragment。

而对于Fragment我们知道，它是可以和Activity的生命周期绑定到一起的，而Fragment有两种表现形式，一种是有界面的Fragment，一种是无界面的Fragment。而在里这个RequestManagerFragment就是一个无界面的Fragment。

Glide巧妙的借助这种不需要界面的Fragment顺利地与Activity或者Fragment的生命周期绑定到一起，进而能够顺理成章的监听Activity或Fragment的声明周期。

接下来，

```
RequestManager requestManager =  
current.getRequestManager();  
if (requestManager == null) {  
    requestManager = new RequestManager(context,  
current.getLifecycle(),  
current.getRequestManagerTreeNode());  
    current.setRequestManager(requestManager);  
}  
return requestManager;
```

current.setRequestManager(requestManager);这句话就将RequestManagerFragment与RequestManager绑定到一起（它们是一对一绑定的关系）。

因此，RequestManager的另一个重要作用就是：监听加载图片界面（Activity、Fragment）的生命周期。

接下来我们简单看一下RequestManagerFragment：

```
public class RequestManagerFragment extends Fragment {  
    private final ActivityFragmentLifecycle lifecycle;  
}
```

```
class ActivityFragmentLifecycle implements Lifecycle {
    void onStart() {
        isStarted = true;
        for (LifecycleListener lifecycleListener :
util.getSnapshot(lifecycleListeners)) {
            lifecycleListener.onStart();
        }
    }

    void onStop() {
        isStarted = false;
        for (LifecycleListener lifecycleListener :
util.getSnapshot(lifecycleListeners)) {
            lifecycleListener.onStop();
        }
    }

    void onDestroy() {
        isDestroyed = true;
        for (LifecycleListener lifecycleListener :
util.getSnapshot(lifecycleListeners)) {
            lifecycleListener.onDestroy();
        }
    }
}
```

3.2 Glide源码学习二：load()详解

RequestManager:

由于with()方法返回的是一个RequestManager对象，那么很容易就能想到，load()方法是在RequestManager类当中的，所以说我们首先要看的就是RequestManager这个类。不过在上一篇文章中我们学过，Glide是支持图片URL字符串、图片本地路径等等加载形式的，因此RequestManager中也有很多个load()方法的重载。但是这里我们不可能

把每个load()方法的重载都看一遍，因此我们就只选其中一个加载图片URL字符串的load()方法来进行研究吧。

RequestManager类的简化代码如下所示：

```
public class RequestManager implements LifecycleListener
{
    ...

    /**
     * Returns a request builder to load the given {@link String}.
     * signature.
     *
     * @see #fromString()
     * @see #load(Object)
     *
     * @param string A file path, or a uri or url handled
     * by {@link com.bumptech.glide.load.model.UriLoader}.
     */
    public DrawableTypeRequest<String> load(String
string) {
        return (DrawableTypeRequest<String>)
fromString().load(string);
    }

    /**
     * Returns a request builder that loads data from
     * {@link String}s using an empty signature.
     *
     * <p>
     * Note - this method caches data using only the
     * given String as the cache key. If the data is a Uri
     * outside of
     *
     * your control, or you otherwise expect the data
     * represented by the given String to change without the
     * String
    }
```

```
*      identifier changing, Consider using
*      {@link GenericRequestBuilder#signature(Key)}
to mixin a signature
*      you create that identifies the data currently
at the given String that will invalidate the cache if
that data
*      changes. Alternatively, using {@link
DiskCacheStrategy#NONE} and/or
*      {@link
DrawableRequestBuilder#skipMemoryCache(boolean)} may be
appropriate.
*  </p>
*
*  @see #from(Class)
*  @see #load(String)
*/
public DrawableTypeRequest<String> fromString() {
    return loadGeneric(String.class);
}

private <T> DrawableTypeRequest<T>
loadGeneric(Class<T> modelClass) {
    ModelLoader<T, InputStream> streamModelLoader =
Glide.buildStreamModelLoader(modelClass, context);
    ModelLoader<T, ParcelFileDescriptor>
fileDescriptorModelLoader =
Glide.buildFileDescriptorModelLoader(modelClass,
context);
    if (modelClass != null && streamModelLoader ==
null && fileDescriptorModelLoader == null) {
        throw new IllegalArgumentException("Unknown
type " + modelClass + ". You must provide a Model of a
type for"
                + " which there is a registered
ModelLoader, if you are using a custom model, you must
first call"
```

```
        + " Glide#register with a  
ModelLoaderFactory for your custom model class");  
    }  
  
    return optionsApplier.apply(  
        new DrawableTypeRequest<T>(modelClass,  
        streamModelLoader, fileDescriptorModelLoader, context,  
        glide, requestTracker, lifecycle,  
        optionsApplier));  
}
```

...

```
}
```

RequestManager类的代码是非常多的，但是经过我这样简化之后，看上去就比较清爽了。在我们只探究加载图片URL字符串这一个load()方法的情况下，那么比较重要的方法就只剩下上述代码中的这三个方法。

那么我们先来看load()方法，这个方法中的逻辑是非常简单的，只有一行代码，就是先调用了fromString()方法，再调用load()方法，然后把传入的图片URL地址传进去。而fromString()方法也极为简单，就是调用了loadGeneric()方法，并且指定参数为String.class，因为load()方法传入的是一个字符串参数。那么看上去，好像主要的工作都是在loadGeneric()方法中进行的了。

其实loadGeneric()方法也没几行代码，这里分别调用了Glide.buildStreamModelLoader()方法和Glide.buildFileDescriptorModelLoader()方法来获得ModelLoader对象。ModelLoader对象是用于加载图片的，而我们给load()方法传入不同类型的参数，这里也会得到不同的ModelLoader对象。不过buildStreamModelLoader()方法内部的逻辑还是蛮复杂的，这里就不展开介绍了，要不然篇幅实在收不住，感兴趣的话你可以自己研究。由于我们刚才传入的参数是String.class，因此最终得到的是StreamStringLoader对象，它是实现了ModelLoader接口的。

最后我们可以看到，loadGeneric()方法是要返回一个DrawableTypeRequest对象的，因此在loadGeneric()方法的最后又去new了一个DrawableTypeRequest对象，然后把刚才获得的ModelLoader对象，还有一大堆杂七杂八的东西都传了进去。具体每个参数的含义和作用就不解释了，我们只看主线流程。

那么这个DrawableTypeRequest的作用是什么呢？我们来看下它的源码，如下所示：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType> implements  
DownloadOptions {  
    private final ModelLoader<ModelType, InputStream>  
streamModelLoader;  
    private final ModelLoader<ModelType,  
ParcelFileDescriptor> fileDescriptorModelLoader;  
    private final RequestManager.OptionsApplier  
optionsApplier;  
  
    private static <A, Z, R> FixedLoadProvider<A,  
ImageVideoWrapper, Z, R> buildProvider(Glide glide,  
                                         ModelLoader<A, InputStream>  
streamModelLoader,  
                                         ModelLoader<A, ParcelFileDescriptor>  
fileDescriptorModelLoader, Class<Z> resourceClass,  
                                         Class<R> transcodedClass,  
                                         ResourceTranscoder<Z, R> transcoder) {  
        if (streamModelLoader == null &&  
fileDescriptorModelLoader == null) {  
            return null;  
        }  
  
        if (transcoder == null) {  
            transcoder =  
glide.buildTranscoder(resourceClass, transcodedClass);  
        }  
    }  
}
```

```
    DataLoadProvider<Imagevideowrapper, Z>
dataLoadProvider =
glide.buildDataProvider(Imagevideowrapper.class,
                      resourceClass);
    ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
                        fileDescriptorModelLoader);
    return new FixedLoadProvider<A,
Imagevideowrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
}

DrawableTypeRequest(Class<ModelType> modelClass,
ModelLoader<ModelType, InputStream> streamModelLoader,
                    ModelLoader<ModelType, ParcelFileDescriptor>
fileDescriptorModelLoader, Context context, Glide glide,
                    RequestTracker requestTracker, Lifecycle
lifecycle, RequestManager.OptionsApplier optionsApplier)
{
    super(context, modelClass,
          buildProvider(glide, streamModelLoader,
fileDescriptorModelLoader, GifBitmapWrapper.class,
                        GlideDrawable.class, null),
          glide, requestTracker, lifecycle);
    this.streamModelLoader = streamModelLoader;
    this.fileDescriptorModelLoader =
fileDescriptorModelLoader;
    this.optionsApplier = optionsApplier;
}

/**
 * Attempts to always load the resource as a {@link
android.graphics.Bitmap}, even if it could actually be
animated.
 *
 * @return A new request builder for loading a {@link
android.graphics.Bitmap}
 */

```

```
public BitmapTypeRequest<ModelType> asBitmap() {
    return optionsApplier.apply(new
        BitmapTypeRequest<ModelType>(this, streamModelLoader,
            fileDescriptorModelLoader,
            optionsApplier));
}

/**
 * Attempts to always load the resource as a {@link
 * com.bumptech.glide.load.resource.gif.GifDrawable}.
 *
 * <p>
 * If the underlying data is not a GIF, this will
 * fail. As a result, this should only be used if the model
 * represents an animated GIF and the caller
 * wants to interact with the GifDrawable directly. Normally
 * using
 * just an {@link DrawableTypeRequest} is
 * sufficient because it will determine whether or
 * not the given data represents an animated GIF
 * and return the appropriate animated or not animated
 * {@link android.graphics.drawable.Drawable}
 * automatically.
 *
 * </p>
 *
 * @return A new request builder for loading a {@link
 * com.bumptech.glide.load.resource.gif.GifDrawable}.
 */
public GifTypeRequest<ModelType> asGif() {
    return optionsApplier.apply(new
        GifTypeRequest<ModelType>(this, streamModelLoader,
            optionsApplier));
}

...
}
```

这个类中的代码本身就不多，我只是稍微做了一点简化。可以看到，最主要的就是它提供了asBitmap()和asGif()这两个方法。这两个方法我们在上一篇文章当中都是学过的，分别是用于强制指定加载静态图片和动态图片。而从源码中可以看出，它们分别又创建了一个BitmapTypeRequest和GifTypeRequest，如果没有进行强制指定的话，那默认就是使用DrawableTypeRequest。

好的，那么我们再回到RequestManager的load()方法中。刚才已经分析过了，fromString()方法会返回一个DrawableTypeRequest对象，接下来会调用这个对象的load()方法，把图片的URL地址传进去。但是我们刚才看到了，DrawableTypeRequest中并没有load()方法，那么很容易就能猜想到，load()方法是在父类当中的。

DrawableTypeRequest的父类是DrawableRequestBuilder，我们来看下这个类的源码：

```
public class DrawableRequestBuilder<ModelType>
    extends GenericRequestBuilder<ModelType,
ImageVideoWrapper, GifBitmapWrapper, GlideDrawable>
    implements BitmapOptions, DrawableOptions {

    DrawableRequestBuilder(Context context,
    Class<ModelType> modelClass,
    LoadProvider<ModelType, ImageVideoWrapper,
    GifBitmapWrapper, GlideDrawable> loadProvider, Glide
    glide,
    RequestTracker requestTracker, Lifecycle
    lifecycle) {
        super(context, modelClass, loadProvider,
    GlideDrawable.class, glide, requestTracker, lifecycle);
        // Default to animating.
        crossFade();
    }

    public DrawableRequestBuilder<ModelType> thumbnail(
        DrawableRequestBuilder<?> thumbnailRequest) {
        super.thumbnail(thumbnailRequest);
        return this;
    }
}
```

```
}

@Override
public DrawableRequestBuilder<ModelType> thumbnail(
    GenericRequestBuilder<?, ?, ?, GlideDrawable>
thumbnailRequest) {
    super.thumbnail(thumbnailRequest);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
thumbnail(float sizeMultiplier) {
    super.thumbnail(sizeMultiplier);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
sizeMultiplier(float sizeMultiplier) {
    super.sizeMultiplier(sizeMultiplier);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
decoder(ResourceDecoder<ImageVideoWrapper,
GifBitmapWrapper> decoder) {
    super.decoder(decoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
cacheDecoder(ResourceDecoder<File, GifBitmapWrapper>
cacheDecoder) {
    super.cacheDecoder(cacheDecoder);
    return this;
}
```

```
}

@Override
public DrawableRequestBuilder<ModelType>
encoder(ResourceEncoder<GifBitmapWrapper> encoder) {
    super.encoder(encoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
priority(Priority priority) {
    super.priority(priority);
    return this;
}

public DrawableRequestBuilder<ModelType>
transform(BitmapTransformation... transformations) {
    return bitmapTransform(transformations);
}

public DrawableRequestBuilder<ModelType> centerCrop()
{
    return transform(glide.getDrawableCenterCrop());
}

public DrawableRequestBuilder<ModelType> fitCenter()
{
    return transform(glide.getDrawableFitCenter());
}

public DrawableRequestBuilder<ModelType>
bitmapTransform(Transformation<Bitmap>...
bitmapTransformations) {
    GifBitmapWrapperTransformation[] transformations
=
```

```
        new
GifBitmapWrapperTransformation[bitmapTransformations.length];
    for (int i = 0; i < bitmapTransformations.length;
i++) {
    transformations[i] = new
GifBitmapWrapperTransformation(glide.getBitmapPool(),
bitmapTransformations[i]);
}
return transform(transformations);
}

@Override
public DrawableRequestBuilder<ModelType>
transform(Transformation<GifBitmapWrapper>...
transformation) {
    super.transform(transformation);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> transcoder(
    ResourceTranscoder<GifBitmapWrapper,
GlideDrawable> transcoder) {
    super.transcoder(transcoder);
    return this;
}

public final DrawableRequestBuilder<ModelType>
crossFade() {
    super.animate(new
DrawableCrossFadeFactory<GlideDrawable>());
    return this;
}

public DrawableRequestBuilder<ModelType>
crossFade(int duration) {
```

```
        super.animate(new
DrawableCrossFadeFactory<GlideDrawable>(duration));
        return this;
    }

    public DrawableRequestBuilder<ModelType>
crossFade(int animationId, int duration) {
        super.animate(new
DrawableCrossFadeFactory<GlideDrawable>(context,
animationId,
                duration));
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
dontAnimate() {
        super.dontAnimate();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
animate(ViewPropertyAnimation.Animator animator) {
        super.animate(animator);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> animate(int
animationId) {
        super.animate(animationId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
placeholder(int resourceId) {
```

```
        super.placeholder(resourceId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
placeholder(Drawable drawable) {
    super.placeholder(drawable);
    return this;
}

    @Override
    public DrawableRequestBuilder<ModelType>
fallback(Drawable drawable) {
    super.fallback(drawable);
    return this;
}

    @Override
    public DrawableRequestBuilder<ModelType> fallback(int
resourceId) {
    super.fallback(resourceId);
    return this;
}

    @Override
    public DrawableRequestBuilder<ModelType> error(int
resourceId) {
    super.error(resourceId);
    return this;
}

    @Override
    public DrawableRequestBuilder<ModelType>
error(Drawable drawable) {
    super.error(drawable);
    return this;
}
```

```
    @Override
    public DrawableRequestBuilder<ModelType> listener(
        RequestListener<? super ModelType,
        GlideDrawable> requestListener) {
        super.listener(requestListener);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
diskCacheStrategy(DiskCacheStrategy strategy) {
        super.diskCacheStrategy(strategy);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
skipMemoryCache(boolean skip) {
        super.skipMemoryCache(skip);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> override(int
width, int height) {
        super.override(width, height);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
sourceEncoder(Encoder<ImageVideoWrapper> sourceEncoder) {
        super.sourceEncoder(sourceEncoder);
        return this;
    }

    @Override
```

```
public DrawableRequestBuilder<ModelType>
dontTransform() {
    super.dontTransform();
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
signature(Key signature) {
    super.signature(signature);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
load(ModelType model) {
    super.load(model);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> clone() {
    return (DrawableRequestBuilder<ModelType>)
super.clone();
}

@Override
public Target<GlideDrawable> into(ImageView view) {
    return super.into(view);
}

@Override
void applyFitCenter() {
    fitCenter();
}

@Override
void applyCenterCrop() {
```

```
        centerCrop();
    }
}
```

DrawableRequestBuilder中有很多个方法，这些方法其实就是Glide绝大多数的API了。里面有不少我们在上篇文章中已经用过了，比如说placeholder()方法、error()方法、diskCacheStrategy()方法、override()方法等。当然还有很多暂时还没用到的API，我们会在后面的文章当中学习。

到这里，第二步load()方法也就分析结束了。为什么呢？因为你会发现DrawableRequestBuilder类中有一个into()方法（上述代码第220行），也就是说，最终load()方法返回的其实就是一个DrawableTypeRequest对象。那么接下来我们就要进行第三步了，分析into()方法中的逻辑。

我么可以看到DrawableRequestBuilder这个类中都是一些基本方法，再进一步看它的父类：

GenericRequestBuilder:

```
public class GenericRequestBuilder<ModelType, DataType,
ResourceType, TranscodeType> implements Cloneable {
    protected final Class<ModelType> modelClass;
    protected final Context context;
    protected final Glide glide;
    protected final Class<TranscodeType> transcodeClass;
    protected final RequestTracker requestTracker;
    protected final Lifecycle lifecycle;
    private ChildLoadProvider<ModelType, DataType,
ResourceType, TranscodeType> loadProvider;

    private ModelType model;
    private Key signature = EmptySignature.obtain();
    // model may occasionally be null, so to enforce that
    load() was called, set a boolean rather than relying on
    model
    // not to be null.
    private boolean isModelSet;
```

```
private int placeholderId;
private int errorId;
private RequestListener<? super ModelType,
TranscodeType> requestListener;
private Float thumbSizeMultiplier;
private GenericRequestBuilder<?, ?, ?, TranscodeType>
thumbnailRequestBuilder;
private Float sizeMultiplier = 1f;
private Drawable placeholderDrawable;
private Drawable errorPlaceholder;
private Priority priority = null;
private boolean isCacheable = true;
private GlideAnimationFactory<TranscodeType>
animationFactory = NoAnimation.getFactory();
private int overrideHeight = -1;
private int overridewidth = -1;
private DiskCacheStrategy diskCacheStrategy =
DiskCacheStrategy.RESULT;
private Transformation<ResourceType> transformation =
UnitTransformation.get();
private boolean isTransformationSet;
private boolean isThumbnailBuilt;
private Drawable fallbackDrawable;
private int fallbackResource;

GenericRequestBuilder<LoadProvider<ModelType,
DataType, ResourceType, TranscodeType> loadProvider,
Class<TranscodeType> transcodeClass,
GenericRequestBuilder<ModelType, ?, ?, ?> other) {
    this(other.context, other.modelClass,
loadProvider, transcodeClass, other.glide,
other.requestTracker,
        other.lifecycle);
    this.model = other.model;
    this.isModelSet = other.isModelSet;
    this.signature = other.signature;
    this.diskCacheStrategy = other.diskCacheStrategy;
    this.isCacheable = other.isCacheable;
```

```
}
```

```
    GenericRequestBuilder(Context context,
Class<ModelType> modelClass,
        LoadProvider<ModelType, DataType,
ResourceType, TranscodeType> loadProvider,
            Class<TranscodeType> transcodeClass, Glide
glide, RequestTracker requestTracker, Lifecycle
lifecycle) {
    this.context = context;
    this.modelClass = modelClass;
    this.transcodeClass = transcodeClass;
    this.glide = glide;
    this.requestTracker = requestTracker;
    this.lifecycle = lifecycle;
    this.loadProvider = loadProvider != null
        ? new ChildLoadProvider<ModelType,
        DataType, ResourceType, TranscodeType>(loadProvider) :
    null;

    if (context == null) {
        throw new NullPointerException("Context can't
be null");
    }
    if (modelClass != null && loadProvider == null) {
        throw new NullPointerException("LoadProvider
must not be null");
    }
}
```

```
    public GenericRequestBuilder<ModelType, DataType,
    ResourceType, TranscodeType> thumbnail(
        GenericRequestBuilder<?, ?, ?, TranscodeType>
    thumbnailRequest) {
        if (this.equals(thumbnailRequest)) {
```

```
        throw new IllegalArgumentException("You  
cannot set a request as a thumbnail for itself. Consider  
using "  
        + "clone() on the request you are  
passing to thumbnail()");  
    }  
    this.thumbnailRequestBuilder = thumbnailRequest;  
  
    return this;  
}  
  
.....  
}
```

可以看到这个GenericRequestBuilder就是我们Glide的一系列设置的最终实现类。它可以通过链式调用来配置一系列的参数。

RequestTracker: 请求跟踪器。这个类在后面的into方法里会经常看到。它的作用就是负责跟踪我们图片request的周期，它可以取消request，也可以重启一些失败的request。

同时我们可以GenericRequestBuilder这里看到我们可以在调用Glide能够设置的一系列参数：

```
private int placeholderId;  
private int errorId;  
private Drawable errorPlaceholder;  
private Priority priority = null;  
private int overrideHeight = -1;  
private int overridewidth = -1;  
private DiskCacheStrategy diskCacheStrategy =  
DiskCacheStrategy.RESULT;  
private Transformation<ResourceType> transformation =  
UnitTransformation.get();  
.....
```

接着看load()方法：

```
public DrawableTypeRequest<String> load(String string) {
```

```
        return (DrawableTypeRequest<String>)
fromString().load(string);
}

public DrawableTypeRequest<String> fromString() {
    return loadGeneric(String.class);
}

private <T> DrawableTypeRequest<T> loadGeneric(Class<T>
modelClass) {
    //这里我们看到了两个ModelLoader，它的作用就是把Model数据
源转为原始数据Data
    ModelLoader<T, InputStream> streamModelLoader =
Glide.buildStreamModelLoader(modelClass, context);
    ModelLoader<T, ParcelFileDescriptor>
fileDescriptorModelLoader =
Glide.buildFileDescriptorModelLoader(modelClass,
context);
    if (modelClass != null && streamModelLoader ==
null && fileDescriptorModelLoader == null) {
        throw new IllegalArgumentException("Unknown
type " + modelClass + ". You must provide a Model of a
type for"
                + " which there is a registered
ModelLoader, if you are using a custom model, you must
first call"
                + " Glide#register with a
ModelLoaderFactory for your custom model class");
    }
    //创建DrawableTypeRequest对象
    return optionsApplier.apply(
        new DrawableTypeRequest<T>(modelClass,
streamModelLoader, fileDescriptorModelLoader, context,
                    glide, requestTracker, lifecycle,
optionsApplier));
}
```

```
@Override
    public DrawableRequestBuilder<ModelType>
load(ModelType model) {
    super.load(model);
    return this;
}

public GenericRequestBuilder<ModelType, DataType,
ResourceType, TranscodeType> load(ModelType model) {
    this.model = model;
    isModelSet = true;
    return this;
}
```

可以看到这段代码的最后创建DrawableTypeRequest对象。

```
//强制加载静态图片
public BitmapTypeRequest<ModelType> asBitmap() {
    return optionsApplier.apply(new
BitmapTypeRequest<ModelType>(this, streamModelLoader,
                                fileDescriptorModelLoader,
optionsApplier));
}

//强制加载动态图片
public GifTypeRequest<ModelType> asGif() {
    return optionsApplier.apply(new
GifTypeRequest<ModelType>(this, streamModelLoader,
optionsApplier));
}
```

而DrawableTypeRequest类中比较重要的两个方法就是asBitmap和asGif，那么可以看出DrawableTypeRequest的主要作用就是把图片按照静态bitmap或者动态gif的形式进行设置。

总结：

load方法最终做的就是进行一些初始化的操作，获得了一个DrawableTypeRequest对象，通过这个对象我们就可以获得图片请求的request，然后再接下来的into方法中使用。

3.3 Glide源码学习三：into()详解

如果说前面两步都是在准备开胃小菜的话，那么现在终于要进入主菜了，因为into()方法也是整个Glide图片加载流程中逻辑最复杂的地方。

不过从刚才的代码来看，into()方法中并没有任何逻辑，只有一句super.into(view)。那么很显然，into()方法的具体逻辑都是在DrawableRequestBuilder的父类当中了。

DrawableRequestBuilder的父类是GenericRequestBuilder，我们来看一下GenericRequestBuilder类中的into()方法，如下所示：

```
public Target<TranscodeType> into(ImageView view) {
    Util.assertMainThread();
    if (view == null) {
        throw new IllegalArgumentException("You must pass
in a non null view");
    }
    if (!isTransformationSet && view.getScaleType() != null) {
        switch (view.getScaleType()) {
            case CENTER_CROP:
                applyCenterCrop();
                break;
            case FIT_CENTER:
            case FIT_START:
            case FIT_END:
                applyFitCenter();
                break;
            // $CASES-OMITTED$
            default:
                // Do nothing.
        }
    }
}
```

```
    }

    return into(glide.buildImageViewTarget(view,
transcodeClass));
}
```

这里前面一大堆的判断逻辑我们都可以先不用管，等到后面文章讲transform的时候会再进行解释，现在我们只需要关注最后一行代码。最后一行代码先是调用了glide.buildImageViewTarget()方法，这个方法会构建出一个Target对象，Target对象则是用来最终展示图片用的，如果我们跟进去的话会看到如下代码：

```
<R> Target<R> buildImageViewTarget(ImageView imageView,
Class<R> transcodedClass) {
    return imageViewTargetFactory.buildTarget(imageView,
transcodedClass);
}
```

这里其实又是调用了ImageViewTargetFactory的buildTarget()方法，我们继续跟进去，代码如下所示：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view,
Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz))
{
            return (Target<Z>) new
GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new
BitmapImageViewTarget(view);
        } else if
(Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new
DrawableImageViewTarget(view);
        } else {
```

```
        throw new IllegalArgumentException("Unhandled
class: " + clazz
                + ", try .as"
(class).transcode(ResourceTranscoder)");
}
}
}
```

可以看到，在buildTarget()方法中会根据传入的class参数来构建不同的Target对象。那如果你要分析这个class参数是从哪儿传过来的，这可有得你分析了，简单起见我直接帮大家梳理清楚。这个class参数其实基本上只有两种情况，如果你在使用Glide加载图片的时候调用了asBitmap()方法，那么这里就会构建出BitmapImageViewTarget对象，否则的话构建的都是GlideDrawableImageViewTarget对象。至于上述代码中的DrawableImageViewTarget对象，这个通常都是用不到的，我们可以暂时不用管它。

也就是说，通过glide.buildImageViewTarget()方法，我们构建出了一个GlideDrawableImageViewTarget对象。那现在回到刚才into()方法的最后一行，可以看到，这里又将这个参数传入到了GenericRequestBuilder另一个接收Target对象的into()方法当中了。我们来看一下这个into()方法的源码：

```
public <Y extends Target<TranscodeType>> Y into(Y target)
{
    Util.assertMainThread();
    if (target == null) {
        throw new IllegalArgumentException("You must pass
in a non null Target");
    }
    if (!isModelSet) {
        throw new IllegalArgumentException("You must
first set a model (try #load())");
    }
    Request previous = target.getRequest();
    if (previous != null) {
        previous.clear();
    }
    target.setListener(listener);
    return target;
}
```

```
    requestTracker.removeRequest(previous);
    previous.recycle();
}

Request request = buildRequest(target);
target.setRequest(request);
lifecycle.addListener(target);
requestTracker.runRequest(request);
return target;
}
```

这里我们还是只抓核心代码，其实只有两行是最关键的，第15行调用buildRequest()方法构建出了一个Request对象，还有第18行来执行这个Request。

Request是用来发出加载图片请求的，它是Glide中非常关键的一个组件。我们先来看buildRequest()方法是如何构建Request对象的：

```
private Request buildRequest(Target<TranscodeType>
target) {
    if (priority == null) {
        priority = Priority.NORMAL;
    }
    return buildRequestRecursive(target, null);
}

private Request
buildRequestRecursive(Target<TranscodeType> target,
ThumbnailRequestCoordinator parentCoordinator) {
    if (thumbnailRequestBuilder != null) {
        if (isThumbnailBuilt) {
            throw new IllegalStateException("You cannot
use a request as both the main request and a thumbnail, "
                + "consider using clone() on the
request(s) passed to thumbnail()");
        }
        // Recursive case: contains a potentially
        recursive thumbnail request builder.
    }
}
```

```
        if  
        (thumbnailRequestBuilder.animationFactory.equals(NoAnimat  
ion.getFactory())) {  
            thumbnailRequestBuilder.animationFactory =  
            animationFactory;  
        }  
  
        if (thumbnailRequestBuilder.priority == null) {  
            thumbnailRequestBuilder.priority =  
            getThumbnailPriority();  
        }  
  
        if (util.isValidDimensions(overrideWidth,  
        overrideHeight)  
            &&  
        !util.isValidDimensions(thumbnailRequestBuilder.overrideW  
idth,  
        thumbnailRequestBuilder.overrideHeight)) {  
            thumbnailRequestBuilder.override(overrideWidth,  
            overrideHeight);  
        }  
  
        ThumbnailRequestCoordinator coordinator = new  
        ThumbnailRequestCoordinator(parentCoordinator);  
        Request fullRequest = obtainRequest(target,  
        sizeMultiplier, priority, coordinator);  
        // Guard against infinite recursion.  
        isThumbnailBuilt = true;  
        // Recursively generate thumbnail requests.  
        Request thumbRequest =  
        thumbnailRequestBuilder.buildRequestRecursive(target,  
        coordinator);  
        isThumbnailBuilt = false;  
        coordinator.setRequests(fullRequest,  
        thumbRequest);  
        return coordinator;  
    } else if (thumbSizeMultiplier != null) {
```

```
// Base case: thumbnail multiplier generates a
// thumbnail request, but cannot recurse.
ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
Request fullRequest = obtainRequest(target,
sizeMultiplier, priority, coordinator);
Request thumbnailRequest = obtainRequest(target,
thumbSizeMultiplier, getThumbnailPriority(),
coordinator);
coordinator.setRequests(fullRequest,
thumbnailRequest);
return coordinator;
} else {
// Base case: no thumbnail.
return obtainRequest(target, sizeMultiplier,
priority, parentCoordinator);
}
}

private Request obtainRequest(Target<TranscodeType>
target, float sizeMultiplier, Priority priority,
RequestCoordinator requestCoordinator) {
return GenericRequest.obtain(
loadProvider,
model,
signature,
context,
priority,
target,
sizeMultiplier,
placeholderDrawable,
placeholderId,
errorPlaceholder,
errorId,
fallbackDrawable,
fallbackResource,
requestListener,
requestCoordinator,
```

```
        glide.getEngine(),
        transformation,
        transcodeClass,
        isCacheable,
        animationFactory,
        overrideWidth,
        overrideHeight,
        diskCacheStrategy);
}
```

可以看到，`buildRequest()`方法的内部其实又调用了`buildRequestRecursive()`方法，而`buildRequestRecursive()`方法中的代码虽然有点长，但是其中90%的代码都是在处理缩略图的。如果我们只追主线流程的话，那么只需要看第47行代码就可以了。这里调用了`obtainRequest()`方法来获取一个`Request`对象，而`obtainRequest()`方法中又去调用了`GenericRequest`的`obtain()`方法。注意这个`obtain()`方法需要传入非常多的参数，而其中很多的参数我们都是比较熟悉的，像什么`placeholderId`、`errorPlaceholder`、`diskCacheStrategy`等等。因此，我们就有理由猜测，刚才在`load()`方法中调用的所有API，其实都是在这里组装到`Request`对象当中的。那么我们进入到这个`GenericRequest`的`obtain()`方法瞧一瞧：

```
public final class GenericRequest<A, T, Z, R> implements
Request, SizeReadyCallback,
ResourceCallback {

    ...

    public static <A, T, Z, R> GenericRequest<A, T, Z, R>
obtain(
        LoadProvider<A, T, Z, R> loadProvider,
        A model,
        Key signature,
        Context context,
        Priority priority,
        Target<R> target,
        float sizeMultiplier,
```

```
        Drawable placeholderDrawable,
        int placeholderResourceId,
        Drawable errorDrawable,
        int errorResourceId,
        Drawable fallbackDrawable,
        int fallbackResourceId,
        RequestListener<? super A, R>
requestListener,
        RequestCoordinator requestCoordinator,
        Engine engine,
        Transformation<Z> transformation,
        Class<R> transcodeClass,
        boolean isMemoryCacheable,
        GlideAnimationFactory<R> animationFactory,
        int overrideWidth,
        int overrideHeight,
        DiskCacheStrategy diskCacheStrategy) {
    @SuppressWarnings("unchecked")
    GenericRequest<A, T, Z, R> request =
(GenericRequest<A, T, Z, R>) REQUEST_POOL.poll();
    if (request == null) {
        request = new GenericRequest<A, T, Z, R>();
    }
    request.init(loadProvider,
            model,
            signature,
            context,
            priority,
            target,
            sizeMultiplier,
            placeholderDrawable,
            placeholderResourceId,
            errorDrawable,
            errorResourceId,
            fallbackDrawable,
            fallbackResourceId,
            requestListener,
            requestCoordinator,
```

```
        engine,
        transformation,
        transcodeClass,
        isMemoryCacheable,
        animationFactory,
        overrideWidth,
        overrideHeight,
        diskCacheStrategy);

    return request;
}

...
}
```

可以看到，这里在第33行去new了一个GenericRequest对象，并在最后一行返回，也就是说，obtain()方法实际上获得的就是一个GenericRequest对象。另外这里又在第35行调用了GenericRequest的init()，里面主要就是一些赋值的代码，将传入的这些参数赋值到GenericRequest的成员变量当中，我们就不再跟进去看了。

好，那现在解决了构建Request对象的问题，接下来我们看一下这个Request对象又是怎么执行的。回到刚才的into()方法，你会发现在第18行调用了requestTracker.runRequest()方法来去执行这个Request，那么我们跟进去瞧一瞧，如下所示：

```
/***
 * Starts tracking the given request.
 */
public void runRequest(Request request) {
    requests.add(request);
    if (!isPaused) {
        request.begin();
    } else {
        pendingRequests.add(request);
    }
}
```

这里有一个简单的逻辑判断，就是先判断Glide当前是不是处理暂停状态，如果不是暂停状态就调用Request的begin()方法来执行Request，否则的话就先将Request添加到待执行队列里面，等暂停状态解除了之后再执行。

暂停请求的功能仍然不是这篇文章所关心的，这里就直接忽略了，我们重点来看这个begin()方法。由于当前的Request对象是一个GenericRequest，因此这里就需要看GenericRequest中的begin()方法了，如下所示：

```
@Override  
public void begin() {  
    startTime = LogTime.getLogTime();  
    if (model == null) {  
        onException(null);  
        return;  
    }  
    status = Status.WAITING_FOR_SIZE;  
    if (util.isValidDimensions(overrideWidth,  
    overrideHeight)) {  
        onSizeReady(overrideWidth, overrideHeight);  
    } else {  
        target.getSize(this);  
    }  
    if (!isComplete() && !isFailed() &&  
    canNotifyStatusChanged()) {  
        target.onLoadStarted(getPlaceholderDrawable());  
    }  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("finished run method in " +  
        LogTime.getElapsedMillis(startTime));  
    }  
}
```

这里我们来注意几个细节，首先如果model等于null，model也就是我们在第二步load()方法中传入的图片URL地址，这个时候会调用onException()方法。如果你跟到onException()方法里面去看看，你会发现它最终会调用到一个setErrorPlaceholder()当中，如下所示：

```
private void setErrorPlaceholder(Exception e) {  
    if (!canNotifyStatusChanged()) {  
        return;  
    }  
    Drawable error = model == null ?  
        getFallbackDrawable() : null;  
    if (error == null) {  
        error = getErrorDrawable();  
    }  
    if (error == null) {  
        error = getPlaceholderDrawable();  
    }  
    target.onLoadFailed(e, error);  
}
```

这个方法中会先去获取一个error的占位图，如果获取不到的话会再去获取一个loading占位图，然后调用target.onLoadFailed()方法并将占位图传入。那么onLoadFailed()方法中做了什么呢？我们看一下：

```
public abstract class ImageViewTarget<Z> extends  
ViewTarget<ImageView, Z> implements  
GlideAnimation.ViewAdapter {  
  
    ...  
  
    @Override  
    public void onLoadStarted(Drawable placeholder) {  
        view.setImageDrawable(placeholder);  
    }  
  
    @Override  
    public void onLoadFailed(Exception e, Drawable  
errorDrawable) {  
        view.setImageDrawable(errorDrawable);  
    }  
  
    ...
```

```
}
```

很简单，其实就是将这张error占位图显示到ImageView上而已，因为现在出现了异常，没办法展示正常的图片了。而如果你仔细看下刚才begin()方法的第15行，你会发现它又调用了一个target.onLoadStarted()方法，并传入了一个loading占位图，在也就说，在图片请求开始之前，会先使用这张占位图代替最终的图片显示。这也是我们在上一篇文章中学过的placeholder()和error()这两个占位图API底层的实现原理。

好，那么我们继续回到begin()方法。刚才讲了占位图的实现，那么具体的图片加载又是从哪里开始的呢？是在begin()方法的第10行和第12行。这里要分两种情况，一种是你使用了override() API为图片指定了一个固定的宽高，一种是没有指定。如果指定了的话，就会执行第10行代码，调用onSizeReady()方法。如果没指定的话，就会执行第12行代码，调用target.getSize()方法。这个target.getSize()方法的内部会根据ImageView的layout_width和layout_height值做一系列的计算，来算出图片应该的宽高。具体的计算细节我就不带着大家分析了，总之在计算完之后，它也会调用onSizeReady()方法。也就是说，不管是哪种情况，最终都会调用到onSizeReady()方法，在这里进行下一步操作。那么我们跟到这个方法里面来：

```
@Override  
public void onSizeReady(int width, int height) {  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("Got onSizeReady in " +  
LogTime.getElapsedMillis(startTime));  
    }  
    if (status != Status.WAITING_FOR_SIZE) {  
        return;  
    }  
    status = Status.RUNNING;  
    width = Math.round(sizeMultiplier * width);  
    height = Math.round(sizeMultiplier * height);  
    ModelLoader<A, T> modelLoader =  
loadProvider.getModelLoader();  
    final DataFetcher<T> dataFetcher =  
modelLoader.getResourceFetcher(model, width, height);
```

```
    if (dataFetcher == null) {
        onException(new Exception("Failed to load model:
\" + model + "\""));
        return;
    }
    ResourceTranscoder<Z, R> transcoder =
loadProvider.getTranscoder();
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished setup for calling load in " +
LogTime.getElapsedMillis(startTime));
    }
    loadedFromMemoryCache = true;
    loadStatus = engine.load(signature, width, height,
dataFetcher, loadProvider, transformation, transcoder,
                    priority, isMemoryCacheable,
diskCacheStrategy, this);
    loadedFromMemoryCache = resource != null;
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished onSizeReady in " +
LogTime.getElapsedMillis(startTime));
    }
}
```

从这里开始，真正复杂的地方来了，我们需要慢慢进行分析。先来看一下，在第12行调用了loadProvider.getModelLoader()方法，那么我们第一个要搞清楚的就是，这个loadProvider是什么？要搞清楚这点，需要先回到第二步的load()方法当中。还记得load()方法是返回一个DrawableTypeRequest对象吗？刚才我们只是分析了DrawableTypeRequest当中的asBitmap()和asGif()方法，并没有仔细看它的构造函数，现在我们重新来看一下DrawableTypeRequest类的构造函数：

```
public class DrawableTypeRequest<ModelType> extends
DrawableRequestBuilder<ModelType> implements
DownloadOptions {
```

```
    private final ModelLoader<ModelType, InputStream>
streamModelLoader;
    private final ModelLoader<ModelType,
ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier
optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A,
ImageVideoWrapper, Z, R> buildProvider(Glide glide,
                                         ModelLoader<A, InputStream>
streamModelLoader,
                                         ModelLoader<A, ParcelFileDescriptor>
fileDescriptorModelLoader, Class<Z> resourceClass,
                                         Class<R> transcodedClass,
                                         ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null &&
fileDescriptorModelLoader == null) {
            return null;
        }
        if (transcoder == null) {
            transcoder =
glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z>
dataLoadProvider =
glide.buildDataProvider(ImageVideoWrapper.class,
                           resourceClass);
        ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
                           fileDescriptorModelLoader);
        return new FixedLoadProvider<A,
ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
    }

    DrawableTypeRequest<Class<ModelType> modelClass,
ModelLoader<ModelType, InputStream> streamModelLoader,
```

```
    ModelLoader<ModelType, ParcelFileDescriptor>
fileDescriptorModelLoader, Context context, Glide glide,
    RequestTracker requestTracker, Lifecycle
lifecycle, RequestManager.OptionsApplier optionsApplier)
{
    super(context, modelClass,
        buildProvider(glide, streamModelLoader,
fileDescriptorModelLoader, GifBitmapWrapper.class,
                GlideDrawable.class, null),
        glide, requestTracker, lifecycle);
    this.streamModelLoader = streamModelLoader;
    this.fileDescriptorModelLoader =
fileDescriptorModelLoader;
    this.optionsApplier = optionsApplier;
}
...
}
```

可以看到，这里在第29行，也就是构造函数中，调用了一个buildProvider()方法，并把streamModelLoader和fileDescriptorModelLoader等参数传入到这个方法中，这两个ModelLoader就是之前在loadGeneric()方法中构建出来的。

那么我们再来看一下buildProvider()方法里面做了什么，在第16行调用了glide.buildTranscoder()方法来构建一个ResourceTranscoder，它是用于对图片进行转码的，由于ResourceTranscoder是一个接口，这里实际会构建出一个GifBitmapWrapperDrawableTranscoder对象。

接下来在第18行调用了glide.buildDataProvider()方法来构建一个DataLoadProvider，它是用于对图片进行编解码的，由于DataLoadProvider是一个接口，这里实际会构建出一个ImageVideoGifDrawableLoadProvider对象。

然后在第20行，new了一个ImageVideoModelLoader的实例，并把之前loadGeneric()方法中构建的两个ModelLoader封装到了ImageVideoModelLoader当中。

最后，在第22行，new出一个FixedLoadProvider，并把刚才构建的出来的GifBitmapWrapperDrawableTranscoder、ImageVideoModelLoader、ImageVideoGifDrawableLoadProvider都封装进去，这个也就是onSizeReady()方法中的loadProvider了。

好的，那么我们回到onSizeReady()方法中，在onSizeReady()方法的第12行和第18行，分别调用了loadProvider的getModelLoader()方法和getTranscoder()方法，那么得到的对象也就是刚才我们分析的ImageVideoModelLoader和GifBitmapWrapperDrawableTranscoder了。而在第13行，又调用了ImageVideoModelLoader的getResourceFetcher()方法，这里我们又需要跟进去瞧一瞧了，代码如下所示：

```
public class ImagevideoModelLoader<A> implements ModelLoader<A, Imagevideowrapper> {
    private static final String TAG = "IVML";

    private final ModelLoader<A, InputStream> streamLoader;
    private final ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader;

    public ImagevideoModelLoader(ModelLoader<A, InputStream> streamLoader,
                                 ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader) {
        if (streamLoader == null && fileDescriptorLoader == null) {
            throw new NullPointerException("At least one of streamLoader and fileDescriptorLoader must be non null");
        }
        this.streamLoader = streamLoader;
        this.fileDescriptorLoader = fileDescriptorLoader;
    }

    @Override
```

```
public DataFetcher<Imagevideowrapper>
getResourceFetcher(A model, int width, int height) {
    DataFetcher<InputStream> streamFetcher = null;
    if (streamLoader != null) {
        streamFetcher =
streamLoader.getResourceFetcher(model, width, height);
    }
    DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher = null;
    if (fileDescriptorLoader != null) {
        fileDescriptorFetcher =
fileDescriptorLoader.getResourceFetcher(model, width,
height);
    }

    if (streamFetcher != null || fileDescriptorFetcher != null) {
        return new ImagevideoFetcher(streamFetcher,
fileDescriptorFetcher);
    } else {
        return null;
    }
}

static class ImagevideoFetcher implements
DataFetcher<Imagevideowrapper> {
    private final DataFetcher<InputStream>
streamFetcher;
    private final DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher;

    public ImagevideoFetcher(DataFetcher<InputStream>
streamFetcher,
                           DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher) {
        this.streamFetcher = streamFetcher;
        this.fileDescriptorFetcher =
fileDescriptorFetcher;
    }
}
```

```
    }  
  
    ...  
}  
}
```

可以看到，在第20行会先调用streamLoader.getResourceFetcher()方法获取一个DataFetcher，而这个streamLoader其实就是在loadGeneric()方法中构建出的StreamStringLoader，调用它的getResourceFetcher()方法会得到一个HttpUrlFetcher对象。然后在第28行new出了一个ImageVideoFetcher对象，并把获得的HttpUrlFetcher对象传进去。也就是说，ImageVideoModelLoader的getResourceFetcher()方法得到的是一个ImageVideoFetcher。

那么我们再次回到onSizeReady()方法，在onSizeReady()方法的第23行，这里将刚才获得的ImageVideoFetcher、GifBitmapWrapperDrawableTranscoder等等一系列的值一起传入到了Engine的load()方法当中。接下来我们就要看一看，这个Engine的load()方法当中，到底做了什么？代码如下所示：

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {  
  
    ...  
  
    public <T, Z, R> LoadStatus load(Key signature, int  
        width, int height, DataFetcher<T> fetcher,  
        DataLoadProvider<T, Z> loadProvider,  
        Transformation<Z> transformation, ResourceTranscoder<Z,  
        R> transcoder,  
        Priority priority, boolean isMemoryCacheable,  
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)  
    {  
        util.assertMainThread();  
        long startTime = LogTime.getLogTime();  
  
        final String id = fetcher.getId();
```

```
        EngineKey key = keyFactory.buildKey(id,
signature, width, height, loadProvider.getCacheDecoder(),
loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
transcoder,
loadProvider.getSourceEncoder());
```

```
        EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);
        if (cached != null) {
            cb.onResourceReady(cached);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from
cache", startTime, key);
            }
            return null;
        }
```

```
        EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
        if (active != null) {
            cb.onResourceReady(active);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from
active resources", startTime, key);
            }
            return null;
        }
```

```
        EngineJob current = jobs.get(key);
        if (current != null) {
            current.addCallback(cb);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Added to existing
load", startTime, key);
            }
            return new LoadStatus(cb, current);
        }
```

```
        EngineJob engineJob = engineJobFactory.build(key,
isMemoryCacheable);

        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T,
Z, R>(key, width, height, fetcher, loadProvider,
transformation,
                transcoder, diskCacheProvider,
diskCacheStrategy, priority);

        EngineRunnable runnable = new
EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load",
startTime, key);
        }
        return new LoadStatus(cb, engineJob);
    }

    ...
}
```

load()方法中的代码虽然有点长，但大多数的代码都是在处理缓存的。关于缓存的内容我们会在下一篇文章当中学习，现在只需要从第45行看起就行。这里构建了一个EngineJob，它的主要作用就是用来开启线程的，为后面的异步加载图片做准备。接下来第46行创建了一个DecodeJob对象，从名字上来看，它好像是用来对图片进行解码的，但实际上它的任务十分繁重，待会我们就知道了。继续往下看，第48行创建了一个EngineRunnable对象，并且在51行调用了EngineJob的start()方法来运行EngineRunnable对象，这实际上就是让EngineRunnable的run()方法在子线程当中执行了。那么我们现在就可以去看看EngineRunnable的run()方法里做了些什么，如下所示：

```
@Override
public void run() {
```

```
if (isCancelled) {
    return;
}
Exception exception = null;
Resource<?> resource = null;
try {
    resource = decode();
} catch (Exception e) {
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        Log.v(TAG, "Exception decoding", e);
    }
    exception = e;
}
if (isCancelled) {
    if (resource != null) {
        resource.recycle();
    }
    return;
}
if (resource == null) {
    onLoadFailed(exception);
} else {
    onLoadComplete(resource);
}
}
```

这个方法中的代码并不多，但我们仍然还是要抓重点。在第9行，这里调用了一个decode()方法，并且这个方法返回了一个Resource对象。看上去所有的逻辑应该都在这个decode()方法执行的了，那我们跟进去瞧一瞧：

```
private Resource<?> decode() throws Exception {
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}
```

decode()方法中又分了两种情况，从缓存当中去decode图片的话就会执行decodeFromCache()，否则的话就执行decodeFromSource()。本篇文章中我们不讨论缓存的情况，那么就直接来看decodeFromSource()方法的代码吧，如下所示：

```
private Resource<?> decodeFromSource() throws Exception {  
    return decodeJob.decodeFromSource();  
}
```

这里又调用了DecodeJob的decodeFromSource()方法。刚才已经说了，DecodeJob的任务十分繁重，我们继续跟进看一看吧：

```
class DecodeJob<A, T, Z> {  
  
    ...  
  
    public Resource<Z> decodeFromSource() throws  
Exception {  
    Resource<T> decoded = decodeSource();  
    return transformEncodeAndTranscode(decoded);  
}  
  
    private Resource<T> decodeSource() throws Exception {  
    Resource<T> decoded = null;  
    try {  
        long startTime = LogTime.getLogTime();  
        final A data = fetcher.loadData(priority);  
        if (Log.isLoggable(TAG, Log.VERBOSE)) {  
            logWithTimeAndKey("Fetched data",  
startTime);  
        }  
        if (isCancelled) {  
            return null;  
        }  
        decoded = decodeFromSourceData(data);  
    } finally {  
        fetcher.cleanup();  
    }
```

```
        return decoded;
    }

    ...
}
```

主要的方法就这些，我都帮大家提取出来了。那么我们先来看一下 decodeFromSource()方法，其实它的工作分为两部，第一步是调用 decodeSource()方法来获得一个Resource对象，第二步是调用 transformEncodeAndTranscode()方法来处理这个Resource对象。

那么我们先来看第一步，decodeSource()方法中的逻辑也并不复杂，首先在第14行调用了fetcher.loadData()方法。那么这个fetcher是什么呢？其实就是刚才在onSizeReady()方法中得到的ImageVideoFetcher对象，这里调用它的 loadData()方法，代码如下所示：

```
@Override
public Imagevideowrapper loadData(Priority priority)
throws Exception {
    InputStream is = null;
    if (streamFetcher != null) {
        try {
            is = streamFetcher.loadData(priority);
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching input
stream, trying ParcelFileDescriptor", e);
            }
            if (fileDescriptorFetcher == null) {
                throw e;
            }
        }
    }
    ParcelFileDescriptor fileDescriptor = null;
    if (fileDescriptorFetcher != null) {
        try {
            fileDescriptor =
fileDescriptorFetcher.loadData(priority);
```

```
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching
ParcelFileDescriptor", e);
            }
            if (is == null) {
                throw e;
            }
        }
    }
    return new Imagevideowrapper(is, fileDescriptor);
}
```

可以看到，在ImageVideoFetcher的loadData()方法的第6行，这里又去调用了streamFetcher.loadData()方法，那么这个streamFetcher是什么呢？自然就是刚才在组装ImageVideoFetcher对象时传进来的HttpUrlFetcher了。因此这里又会去调用HttpUrlFetcher的loadData()方法，那么我们继续跟进去瞧一瞧：

```
public class HttpUrlFetcher implements
DataFetcher<InputStream> {

    ...
    @Override
    public InputStream loadData(Priority priority) throws
Exception {
        return loadDataWithRedirects(glideurl.toURL(), 0
/*redirects*/, null /*lastUrl*/, glideurl.getHeaders());
    }

    private InputStream loadDataWithRedirects(URL url,
int redirects, URL lastUrl, Map<String, String> headers)
        throws IOException {
        if (redirects >= MAXIMUM_REDIRECTS) {
            throw new IOException("Too many (> " +
MAXIMUM_REDIRECTS + ") redirects!");
        } else {

```

```
// Comparing the URLs using .equals performs
additional network I/O and is generally broken.
// See
http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html.
try {
    if (lastUrl != null &&
url.toURI().equals(lastUrl.toURI())) {
        throw new IOException("In re-direct
loop");
    }
} catch (URISyntaxException e) {
    // Do nothing, this is best effort.
}
urlConnection = connectionFactory.build(url);
for (Map.Entry<String, String> headerEntry :
headers.entrySet()) {

urlConnection.addRequestProperty(headerEntry.getKey(),
headerEntry.getValue());
}
urlConnection.setConnectTimeout(2500);
urlConnection.setReadTimeout(2500);
urlConnection.setUseCaches(false);
urlConnection.setDoInput(true);

// Connect explicitly to avoid errors in decoders
if connection fails.
urlConnection.connect();
if (isCancelled) {
    return null;
}
final int statusCode =
urlConnection.getResponseCode();
if (statusCode / 100 == 2) {
    return
getStreamForSuccessfulRequest(urlConnection);
```

```
        } else if (statusCode / 100 == 3) {
            String redirectUrlString =
urlConnection.getHeaderField("Location");
            if (TextUtils.isEmpty(redirectUrlString)) {
                throw new IOException("Received empty or
null redirect url");
            }
            URL redirectUrl = new URL(url,
redirectUrlString);
            return loadDataWithRedirects(redirectUrl,
redirects + 1, url, headers);
        } else {
            if (statusCode == -1) {
                throw new IOException("Unable to retrieve
response code from HttpURLConnection.");
            }
            throw new IOException("Request failed " +
statusCode + ": " + urlConnection.getResponseMessage());
        }
    }

    private InputStream
getStreamForSuccessfulRequest(HttpURLConnection
urlConnection)
        throws IOException {
    if
(TextUtils.isEmpty(urlConnection.getContentEncoding())) {
        int contentLength =
urlConnection.getContentLength();
        stream =
ContentLengthInputStream.obtain(urlConnection.getInputStream(),
contentLength);
    } else {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Got non empty content
encoding: " + urlConnection.getContentEncoding());
        }
        stream = urlConnection.getInputStream();
    }
}
```

```
    }  
    return stream;  
}  
  
...  
}
```

经过一层一层地跋山涉水，我们终于在这里找到网络通讯的代码了！之前有朋友跟我讲过，说Glide的源码实在是太复杂了，甚至连网络请求是在哪里发出去的都找不到。我们也是经过一段一段又一段的代码跟踪，终于把网络请求的代码给找出来了，实在是太不容易了。

不过也别高兴得太早，现在离最终分析完还早着呢。可以看到，`loadData()`方法只是返回了一个`InputStream`，服务器返回的数据连读都还没开始读呢。所以我们还是要静下心来继续分析，回到刚才`ImageVideoFetcher`的`loadData()`方法中，在这个方法的最后一行，创建了一个`ImageVideoWrapper`对象，并把刚才得到的`InputStream`作为参数传了进去。

然后我们回到再上一层，也就是`DecodeJob`的`decodeSource()`方法当中，在得到了这个`ImageVideoWrapper`对象之后，紧接着又将这个对象传入到了`decodeFromSourceData()`当中，来去解码这个对象。`decodeFromSourceData()`方法的代码如下所示：

```
private Resource<T> decodeFromSourceData(A data) throws  
IOException {  
    final Resource<T> decoded;  
    if (diskCacheStrategy.cacheSource()) {  
        decoded = cacheAndDecodeSourceData(data);  
    } else {  
        long startTime = LogTime.getLogTime();  
        decoded =  
loadProvider.getSourceDecoder().decode(data, width,  
height);  
        if (Log.isLoggable(TAG, Log.VERBOSE)) {  
            logWithTimeAndKey("Decoded from source",  
startTime);  
        }  
    }  
    return decoded;  
}
```

可以看到，这里在第7行调用了
loadProvider.getSourceDecoder().decode()方法来进行解码。
loadProvider就是刚才在onSizeReady()方法中得到的
FixedLoadProvider，而getSourceDecoder()得到的则是一个
GifBitmapWrapperResourceDecoder对象，也就是要调用这个对象的
decode()方法来对图片进行解码。那么我们来看下
GifBitmapWrapperResourceDecoder的代码：

```
public class GifBitmapWrapperResourceDecoder implements  
ResourceDecoder<Imagevideowrapper, GifBitmapwrapper> {  
  
    ...  
  
    @SuppressWarnings("resource")  
    // @see ResourceDecoder.decode  
    @Override  
    public Resource<GifBitmapwrapper>  
decode(Imagevideowrapper source, int width, int height)  
throws IOException {
```

```
        ByteArrayPool pool = ByteArrayPool.get();
        byte[] tempBytes = pool.getBytes();
        GifBitmapwrapper wrapper = null;
        try {
            wrapper = decode(source, width, height,
tempBytes);
        } finally {
            pool.releaseBytes(tempBytes);
        }
        return wrapper != null ? new
GifBitmapwrapperResource(wrapper) : null;
    }

    private GifBitmapwrapper decode(Imagevideowrapper
source, int width, int height, byte[] bytes) throws
IOException {
    final GifBitmapwrapper result;
    if (source.getStream() != null) {
        result = decodeStream(source, width, height,
bytes);
    } else {
        result = decodeBitmapwrapper(source, width,
height);
    }
    return result;
}

private GifBitmapwrapper
decodeStream(Imagevideowrapper source, int width, int
height, byte[] bytes)
    throws IOException {
    InputStream bis =
streamFactory.build(source.getStream(), bytes);
    bis.mark(MARK_LIMIT_BYTES);
    ImageHeaderParser.ImageType type =
parser.parse(bis);
    bis.reset();
    GifBitmapwrapper result = null;
```

```
        if (type == ImageHeaderParser.ImageType.GIF) {
            result = decodeGifwrapper(bis, width,
height);
        }
        // Decoding the gif may fail even if the type
matches.

        if (result == null) {
            // we can only reset the buffered
InputStream, so to start from the beginning of the
stream, we need to

            // pass in a new source containing the
buffered stream rather than the original stream.

            Imagevideowrapper forBitmapDecoder = new
Imagevideowrapper(bis, source.getFileDescriptor());
            result =
decodeBitmapwrapper(forBitmapDecoder, width, height);
        }
        return result;
    }

    private GifBitmapwrapper
decodeBitmapwrapper(Imagevideowrapper toDecode, int
width, int height) throws IOException {
    GifBitmapwrapper result = null;
    Resource<Bitmap> bitmapResource =
bitmapDecoder.decode(toDecode, width, height);
    if (bitmapResource != null) {
        result = new GifBitmapwrapper(bitmapResource,
null);
    }
    return result;
}

...
}
```

首先，在decode()方法中，又去调用了另外一个decode()方法的重载。然后在第23行调用了decodeStream()方法，准备从服务器返回的流当中读取数据。decodeStream()方法中会先从流中读取2个字节的数据，来判断这张图是GIF图还是普通的静图，如果是GIF图就调用decodeGifWrapper()方法来进行解码，如果是普通的静图就用调用decodeBitmapWrapper()方法来进行解码。这里我们只分析普通静图的实现流程，GIF图的实现有点过于复杂了，无法在本篇文章当中分析。

然后我们来看一下decodeBitmapWrapper()方法，这里在第52行调用了bitmapDecoder.decode()方法。这个bitmapDecoder是一个ImageVideoBitmapDecoder对象，那么我们来看一下它的代码，如下所示：

```
public class ImagevideoBitmapDecoder implements
ResourceDecoder<ImageVideoWrapper, Bitmap> {
    private final ResourceDecoder<InputStream, Bitmap>
streamDecoder;
    private final ResourceDecoder<ParcelFileDescriptor,
Bitmap> fileDescriptorDecoder;

    public
ImagevideoBitmapDecoder(ResourceDecoder<InputStream,
Bitmap> streamDecoder,
                    ResourceDecoder<ParcelFileDescriptor, Bitmap>
fileDescriptorDecoder) {
        this.streamDecoder = streamDecoder;
        this.fileDescriptorDecoder =
fileDescriptorDecoder;
    }

    @Override
    public Resource<Bitmap> decode(ImageVideoWrapper
source, int width, int height) throws IOException {
        Resource<Bitmap> result = null;
        InputStream is = source.getStream();
        if (is != null) {
            try {
```

```
        result = streamDecoder.decode(is, width,
height);
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Failed to load image from
stream, trying FileDescriptor", e);
        }
    }
    if (result == null) {
        ParcelFileDescriptor fileDescriptor =
source.getFileDescriptor();
        if (fileDescriptor != null) {
            result =
fileDescriptorDecoder.decode(fileDescriptor, width,
height);
        }
    }
    return result;
}
...
}
```

代码并不复杂，在第14行先调用了source.getStream()来获取到服务器返回的InputStream，然后在第17行调用streamDecoder.decode()方法进行解码。streamDecode是一个StreamBitmapDecoder对象，那么我们再来看这个类的源码，如下所示：

```
public class StreamBitmapDecoder implements
ResourceDecoder<InputStream, Bitmap> {

    ...
    private final Downampler downampler;
    private BitmapPool bitmapPool;
    private DecodeFormat decodeFormat;
```

```
public StreamBitmapDecoder(Downsampler downsample, BitmapPool bitmapPool, DecodeFormat decodeFormat) {  
    this.downsample = downsample;  
    this.bitmapPool = bitmapPool;  
    this.decodeFormat = decodeFormat;  
}  
  
@Override  
public Resource<Bitmap> decode(InputStream source,  
int width, int height) {  
    Bitmap bitmap = downsample.decode(source,  
    bitmapPool, width, height, decodeFormat);  
    return BitmapResource.obtain(bitmap, bitmapPool);  
}  
  
...  
}  
}
```

可以看到，它的decode()方法又去调用了Downsampler的decode()方法。接下来又到了激动人心的时刻了，Downsampler的代码如下所示：

```
public abstract class Downsampler implements  
BitmapDecoder<InputStream> {  
  
...  
  
@Override  
public Bitmap decode(InputStream is, BitmapPool pool,  
int outwidth, int outHeight, DecodeFormat decodeFormat) {  
    final ByteArrayPool byteArrayPool =  
    ByteArrayPool.get();  
    final byte[] bytesForOptions =  
    byteArrayPool.getBytes();  
    final byte[] bytesForStream =  
    byteArrayPool.getBytes();  
    final BitmapFactory.Options options =  
    getDefaultOptions();
```

```
// Use to fix the mark limit to avoid allocating
buffers that fit entire images.

RecyclableBufferedInputStream bufferedStream =
new RecyclableBufferedInputStream(
    is, bytesForStream);

// Use to retrieve exceptions thrown while
reading.

// TODO(#126): when the framework no longer
returns partially decoded Bitmaps or provides a way to
determine

// if a Bitmap is partially decoded, consider
removing.

ExceptionCatchingInputStream exceptionStream =
    ExceptionCatchingInputStream.obtain(bufferedStream);

// Use to read data.

// Ensures that we can always reset after reading
an image header so that we can still attempt to decode
the

// full image even when the header decode fails
and/or overflows our read buffer. See #283.

MarkEnforcingInputStream invalidatingStream = new
MarkEnforcingInputStream(exceptionStream);

try {
    exceptionStream.mark(MARK_POSITION);
    int orientation = 0;
    try {
        orientation = new
ImageHeaderParser(exceptionStream).getOrientation();
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.WARN)) {
            Log.w(TAG, "Cannot determine the
image orientation from header", e);
        }
    } finally {
        try {
            exceptionStream.reset();
        } catch (IOException e) {
```

```
        if (Log.isLoggable(TAG, Log.WARN)) {
            Log.w(TAG, "Cannot reset the
input stream", e);
        }
    }
    options.inTempStorage = bytesForOptions;
    final int[] inDimens =
getDimensions(invalidatingStream, bufferedStream,
options);
    final int inwidth = inDimens[0];
    final int inHeight = inDimens[1];
    final int degreesToRotate =
TransformationUtils.getExifOrientationDegrees(orientation
);
    final int sampleSize =
getRoundedSampleSize(degreesToRotate, inwidth, inHeight,
outwidth, outHeight);
    final Bitmap downsampled =
downsampleWithSize(invalidatingStream, bufferedStream,
options, pool, inwidth, inHeight, sampleSize,
decodeFormat);
    // BitmapFactory swallows exceptions during
decodes and in some cases when inBitmap is non null, may
catch
    // and log a stack trace but still return a
non null bitmap. To avoid displaying partially decoded
bitmaps,
    // we catch exceptions reading from the
stream in our ExceptionCatchingInputStream and throw them
here.
    final Exception streamException =
exceptionStream.getException();
    if (streamException != null) {
        throw new
RuntimeException(streamException);
    }
}
```

```
        Bitmap rotated = null;
        if (downsampled != null) {
            rotated =
TransformationUtils.rotateImageExif(downsampled, pool,
orientation);
            if (!downsampled.equals(rotated) &&
!pool.put(downsampled)) {
                downsampled.recycle();
            }
        }
        return rotated;
    } finally {
        byteArrayPool.releaseBytes(bytesForOptions);
        byteArrayPool.releaseBytes(bytesForStream);
        exceptionStream.release();
        releaseOptions(options);
    }
}

private Bitmap
downsamplewithSize(MarkEnforcingInputStream is,
RecyclableBufferedInputStream bufferedStream,
        BitmapFactory.Options options, BitmapPool
pool, int inwidth, int inHeight, int sampleSize,
        DecodeFormat decodeFormat) {
    // Prior to KitKat, the inBitmap size must
    exactly match the size of the bitmap we're decoding.
    Bitmap.Config config = getConfig(is,
decodeFormat);
    options.inSampleSize = sampleSize;
    options.inPreferredConfig = config;
    if ((options.inSampleSize == 1 ||
Build.VERSION_CODES.KITKAT <= Build.VERSION.SDK_INT) &&
shouldUsePool(is)) {
        int targetwidth = (int) Math.ceil(inwidth /
(double) sampleSize);
        int targetHeight = (int) Math.ceil(inHeight /
(double) sampleSize);
    }
}
```

```
// BitmapFactory will clear out the Bitmap  
before writing to it, so getDirty is safe.  
        setInBitmap(options,  
pool.getDirty(targetwidth, targetHeight, config));  
    }  
    return decodeStream(is, bufferedStream, options);  
}  
  
/**  
 * A method for getting the dimensions of an image  
from the given InputStream.  
*  
* @param is The InputStream representing the image.  
* @param options The options to pass to  
* {@link  
BitmapFactory#decodeStream(InputStream,  
android.graphics.Rect,  
* BitmapFactory.Options)}.  
* @return an array containing the dimensions of the  
image in the form {width, height}.  
*/  
public int[] getDimensions(MarkEnforcingInputStream  
is, RecyclableBufferedInputStream bufferedStream,  
    BitmapFactory.Options options) {  
    options.inJustDecodeBounds = true;  
    decodeStream(is, bufferedStream, options);  
    options.inJustDecodeBounds = false;  
    return new int[] { options.outWidth,  
options.outHeight };  
}  
  
private static Bitmap  
decodeStream(MarkEnforcingInputStream is,  
RecyclableBufferedInputStream bufferedStream,  
    BitmapFactory.Options options) {  
    if (options.inJustDecodeBounds) {
```

```
// This is large, but jpeg headers are not  
size bounded so we need something large enough to  
minimize  
    // the possibility of not being able to fit  
enough of the header in the buffer to get the image size  
so  
    // that we don't fail to load images. The  
BufferedInputStream will create a new buffer of 2x the  
    // original size each time we use up the  
buffer space without passing the mark so this is a  
maximum  
    // bound on the buffer size, not a default.  
Most of the time we won't go past our pre-allocated 16kb.  
    is.mark(MARK_POSITION);  
} else {  
    // Once we've read the image header, we no  
longer need to allow the buffer to expand in size. To  
avoid  
    // unnecessary allocations reading image  
data, we fix the mark limit so that it is no larger than  
our  
    // current buffer size here. See issue #225.  
    bufferedStream.fixMarkLimit();  
}  
final Bitmap result =  
BitmapFactory.decodeStream(is, null, options);  
try {  
    if (options.inJustDecodeBounds) {  
        is.reset();  
    }  
} catch (IOException e) {  
    if (Log.isLoggable(TAG, Log.ERROR)) {  
        Log.e(TAG, "Exception loading  
inDecodeBounds=" + options.inJustDecodeBounds  
                + " sample=" +  
options.inSampleSize, e);  
    }  
}
```

```
        return result;
    }

    ...
}
```

可以看到，对服务器返回的InputStream的读取，以及对图片的加载全都在这里了。当然这里其实处理了很多的逻辑，包括对图片的压缩，甚至还有旋转、圆角等逻辑处理，但是我们目前只需要关注主线逻辑就行了。decode()方法执行之后，会返回一个Bitmap对象，那么图片在这里其实也就已经被加载出来了，剩下的工作就是如果让这个Bitmap显示到界面上，我们继续往下分析。

回到刚才的StreamBitmapDecoder当中，你会发现，它的decode()方法返回的是一个Resource对象。而我们从Downsampler中得到的是一个Bitmap对象，因此这里在第18行又调用了BitmapResource.obtain()方法，将Bitmap对象包装成了Resource对象。代码如下所示：

```
public class BitmapResource implements Resource<Bitmap> {
    private final Bitmap bitmap;
    private final BitmapPool bitmapPool;

    /**
     * Returns a new {@link BitmapResource} wrapping the given {@link Bitmap}
     * if the Bitmap is non-null or null if the
     * given Bitmap is null.
     *
     * @param bitmap A Bitmap.
     * @param bitmapPool A non-null {@link BitmapPool}.
     */
    public static BitmapResource obtain(Bitmap bitmap,
                                       BitmapPool bitmapPool) {
        if (bitmap == null) {
            return null;
        } else {
```

```
        return new BitmapResource(bitmap,
bitmapPool);
    }

    public BitmapResource(Bitmap bitmap, BitmapPool
bitmapPool) {
    if (bitmap == null) {
        throw new NullPointerException("Bitmap must
not be null");
    }
    if (bitmapPool == null) {
        throw new NullPointerException("BitmapPool
must not be null");
    }
    this.bitmap = bitmap;
    this.bitmapPool = bitmapPool;
}

@Override
public Bitmap get() {
    return bitmap;
}

@Override
public int getSize() {
    return Util.getBitmapBytesize(bitmap);
}

@Override
public void recycle() {
    if (!bitmapPool.put(bitmap)) {
        bitmap.recycle();
    }
}
```

BitmapResource的源码也非常简单，经过这样一层包装之后，如果我还需要获取Bitmap，只需要调用Resource的get()方法就可以了。

然后我们需要一层层继续向上返回，StreamBitmapDecoder会将值返回到ImageVideoBitmapDecoder当中，而ImageVideoBitmapDecoder又会将值返回到GifBitmapWrapperResourceDecoder的decodeBitmapWrapper()方法当中。由于代码隔得有点太远了，我重新把decodeBitmapWrapper()方法的代码贴一下：

```
private GifBitmapwrapper  
decodeBitmapwrapper(Imagevideowrapper toDecode, int  
width, int height) throws IOException {  
    GifBitmapwrapper result = null;  
    Resource<Bitmap> bitmapResource =  
    bitmapDecoder.decode(toDecode, width, height);  
    if (bitmapResource != null) {  
        result = new GifBitmapwrapper(bitmapResource,  
null);  
    }  
    return result;  
}
```

可以看到，decodeBitmapWrapper()方法返回的是一个GifBitmapWrapper对象。因此，这里在第5行，又将Resource封装到了一个GifBitmapWrapper对象当中。这个GifBitmapWrapper顾名思义，就是既能封装GIF，又能封装Bitmap，从而保证了不管是什类型的图片Glide都能从容应对。我们顺便来看下GifBitmapWrapper的源码吧，如下所示：

```
public class GifBitmapwrapper {  
    private final Resource<GifDrawable> gifResource;  
    private final Resource<Bitmap> bitmapResource;  
  
    public GifBitmapwrapper(Resource<Bitmap>  
    bitmapResource, Resource<GifDrawable> gifResource) {  
        if (bitmapResource != null && gifResource !=  
null) {
```

```
        throw new IllegalArgumentException("Can only
contain either a bitmap resource or a gif resource, not
both");
    }
    if (bitmapResource == null && gifResource ==
null) {
        throw new IllegalArgumentException("Must
contain either a bitmap resource or a gif resource");
    }
    this.bitmapResource = bitmapResource;
    this.gifResource = gifResource;
}

/**
 * Returns the size of the wrapped resource.
 */
public int getSize() {
    if (bitmapResource != null) {
        return bitmapResource.getSize();
    } else {
        return gifResource.getSize();
    }
}

/**
 * Returns the wrapped {@link Bitmap} resource if it
exists, or null.
*/
public Resource<Bitmap> getBitmapResource() {
    return bitmapResource;
}

/**
 * Returns the wrapped {@link GifDrawable} resource
if it exists, or null.
*/
public Resource<GifDrawable> getGifResource() {
    return gifResource;
}
```

```
    }  
}
```

还是比较简单的，就是分别对gifResource和bitmapResource做了一层封装而已，相信没有什么解释的必要。

然后这个GifBitmapWrapper对象会一直向上返回，返回到GifBitmapWrapperResourceDecoder最外层的decode()方法的时候，会对它再做一次封装，如下所示：

```
@Override  
public Resource<GifBitmapWrapper>  
decode(Imagevideowrapper source, int width, int height)  
throws IOException {  
    ByteArrayPool pool = ByteArrayPool.get();  
    byte[] tempBytes = pool.getBytes();  
    GifBitmapwrapper wrapper = null;  
    try {  
        wrapper = decode(source, width, height,  
tempBytes);  
    } finally {  
        pool.releaseBytes(tempBytes);  
    }  
    return wrapper != null ? new  
GifBitmapwrapperResource(wrapper) : null;  
}
```

3.4 Glide源码学习四：缓存

Glide缓存简介

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就分别来分析一下这两种缓存的使用方法以及它们的实现原理。

缓存Key

既然是缓存功能，就必然会有用于进行缓存的Key。那么Glide的缓存Key是怎么生成的呢？我不得不说，Glide的缓存Key生成规则非常繁琐，决定缓存Key的参数竟然有10个之多。不过繁琐归繁琐，至少逻辑还是比较简单的，我们先来看一下Glide缓存Key的生成逻辑。

生成缓存Key的代码在Engine类的load()方法当中，这部分代码我们在上一篇文章当中已经分析过了，只不过当时忽略了缓存相关的内容，那么我们现在重新来看一下：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    public <T, Z, R> LoadStatus load(Key signature, int
        width, int height, DataFetcher<T> fetcher,
        DataLoadProvider<T, Z> loadProvider,
        Transformation<Z> transformation, ResourceTranscoder<Z,
        R> transcoder,
        Priority priority, boolean isMemoryCacheable,
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)
    {
        Util.assertMainThread();
        long startTime = LogTime.getLogTime();

        final String id = fetcher.getId();
        EngineKey key = keyFactory.buildKey(id,
            signature, width, height, loadProvider.getCacheDecoder(),
```

```
        loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
transcoder,
loadProvider.getSourceEncoder());
}

...
}

}
```

可以看到，这里在第11行调用了fetcher.getId()方法获得了一个id字符串，这个字符串也就是我们要加载的图片的唯一标识，比如说如果是一张网络上的图片的话，那么这个id就是这张图片的url地址。

接下来在第12行，将这个id连同着signature、width、height等等10个参数一起传入到EngineKeyFactory的buildKey()方法当中，从而构建出了一个EngineKey对象，这个EngineKey也就是Glide中的缓存Key了。

可见，决定缓存Key的条件非常多，即使你用override()方法改变了一下图片的width或者height，也会生成一个完全不同的缓存Key。

EngineKey类的源码大家有兴趣可以自己去看一下，其实主要就是重写了equals()和hashCode()方法，保证只有传入EngineKey的所有参数都相同的情况下才认为是同一个EngineKey对象，我就不在这里将源码贴出来了。

内存缓存：

详细讲解：

1、LruCache算法

LruCache算法，又称为近期最少使用算法。主要算法原理就是把最近所使用的对象的强引用存储在LinkedHashMap上，并且，把最近最少使用的对象在缓存池达到预设值之前从内存中移除。

2、弱引用

有了缓存Key，接下来就可以开始进行缓存了，那么我们先从内存缓存看起。

首先你要知道，默认情况下，Glide自动就是开启内存缓存的。也就是说，当我们使用Glide加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用Glide再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个RecyclerView当中反复上下滑动，RecyclerView中只要是Glide加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而Glide最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为Glide默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide对此提供了接口：

```
Glide.with(this)
    .load(url)
    .skipMemoryCache(true)
    .into(imageview);
```

可以看到，只需要调用skipMemoryCache()方法并传入true，就表示禁用掉Glide的内存缓存功能。

没错，关于Glide内存缓存的用法就只有这么多，可以说是相当简单。但是我们不可能只停留在这么简单的层面上，接下来就让我们就通过阅读源码来分析一下Glide的内存缓存功能是如何实现的。

其实说到内存缓存的实现，非常容易就让人想到LruCache算法（Least Recently Used），也叫近期最少使用算法。它的主要算法原理就是把最近使用的对象用强引用存储在LinkedHashMap中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。

那么不必多说，Glide内存缓存的实现自然也是使用的LruCache算法。不过除了LruCache算法之外，Glide还结合了一种弱引用的机制，共同完成了内存缓存功能，下面就让我们来通过源码分析一下。

首先回忆一下，在上一篇文章的第二步load()方法中，我们当时分析到了在loadGeneric()方法中会调用Glide.buildStreamModelLoader()方法来获取一个ModelLoader对象。当时没有再跟进到这个方法的里面再去分析，那么我们现在来看下它的源码：

```
public class Glide {

    public static <T, Y> ModelLoader<T, Y>
    buildModelLoader(Class<T> modelClass, Class<Y>
    resourceClass,
        Context context) {
        if (modelClass == null) {
            if (Log.isLoggable(TAG, Log.DEBUG)) {
                Log.d(TAG, "Unable to load null model,
setting placeholder only");
            }
            return null;
        }
        return
    Glide.get(context).getLoaderFactory().buildModelLoader(mo
    delClass, resourceClass);
    }

    public static Glide get(Context context) {
        if (glide == null) {
            synchronized (Glide.class) {
                if (glide == null) {
                    Context applicationContext =
context.getApplicationContext();
                    List<GlideModule> modules = new
ManifestParser(applicationContext).parse();
                    GlideBuilder builder = new
GlideBuilder(applicationContext);
                    for (GlideModule module : modules) {

module.applyOptions(applicationContext, builder);
                }
                glide = builder.createGlide();
            }
        }
        return glide;
    }
}
```

```
        for (GlideModule module : modules) {  
  
            module.registerComponents(applicationContext, glide);  
        }  
    }  
    return glide;  
}  
  
...  
}
```

这里我们还是只看关键，在第11行去构建ModelLoader对象的时候，先调用了一个Glide.get()方法，而这个方法就是关键。我们可以看到，get()方法中实现的是一个单例功能，而创建Glide对象则是在第24行调用GlideBuilder的createGlide()方法来创建的，那么我们跟到这个方法当中：

```
public class GlideBuilder {  
    ...  
  
    Glide createGlide() {  
        if (sourceService == null) {  
            final int cores = Math.max(1,  
Runtime.getRuntime().availableProcessors());  
            sourceService = new  
FifoPriorityThreadPoolExecutor(cores);  
        }  
        if (diskCacheService == null) {  
            diskCacheService = new  
FifoPriorityThreadPoolExecutor(1);  
        }  
        MemorySizeCalculator calculator = new  
MemorySizeCalculator(context);  
        if (bitmapPool == null) {  
            if (Build.VERSION.SDK_INT >=  
Build.VERSION_CODES.HONEYCOMB) {
```

```
        int size =
calculator.getBitmapPoolSize();
        bitmapPool = new LruBitmapPool(size);
    } else {
        bitmapPool = new BitmapPoolAdapter();
    }
}
if (memoryCache == null) {
    memoryCache = new
LruResourceCache(calculator.getMemoryCachesize());
}
if (diskCacheFactory == null) {
    diskCacheFactory = new
InternalCacheDiskCacheFactory(context);
}
if (engine == null) {
    engine = new Engine(memoryCache,
diskCacheFactory, diskCacheService, sourceService);
}
if (decodeFormat == null) {
    decodeFormat = DecodeFormat.DEFAULT;
}
return new Glide(engine, memoryCache, bitmapPool,
context, decodeFormat);
}
}
```

这里也就是构建Glide对象的地方了。那么观察第22行，你会发现这里new出了一个LruResourceCache，并把它赋值到了memoryCache这个对象上面。你没有猜错，这个就是Glide实现内存缓存所使用的LruCache对象了。不过我这里并不打算展开来讲LruCache算法的具体实现，如果你感兴趣的话可以自己研究一下它的源码。

现在创建好了LruResourceCache对象只能说是要把准备工作做好了，接下来我们就一步步研究Glide中的内存缓存到底是如何实现的。

刚才在Engine的load()方法中我们已经看到了生成缓存Key的代码，而内存缓存的代码其实也是在这里实现的，那么我们重新来看一下Engine类load()方法的完整源码：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    ...

    public <T, Z, R> LoadStatus load(Key signature, int
width, int height, DataFetcher<T> fetcher,
        DataLoadProvider<T, Z> loadProvider,
Transformation<Z> transformation, ResourceTranscoder<Z,
R> transcoder,
        Priority priority, boolean isMemoryCacheable,
DiskCacheStrategy diskCacheStrategy, Resourcecallback cb)
{
    Util.assertMainThread();
    long startTime = LogTime.getLogTime();

    final String id = fetcher.getId();
    EngineKey key = keyFactory.buildKey(id,
signature, width, height, loadProvider.getCacheDecoder(),
        loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
        transcoder,
loadProvider.getSourceEncoder());

    EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);
    if (cached != null) {
        cb.onResourceReady(cached);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Loaded resource from
cache", startTime, key);
        }
        return null;
    }
}
```

```
        EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
        if (active != null) {
            cb.onResourceReady(active);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Loaded resource from
active resources", startTime, key);
            }
            return null;
        }

        EngineJob current = jobs.get(key);
        if (current != null) {
            current.addCallback(cb);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                logWithTimeAndKey("Added to existing
load", startTime, key);
            }
            return new LoadStatus(cb, current);
        }

        EngineJob engineJob = engineJobFactory.build(key,
isMemoryCacheable);
        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T,
Z, R>(key, width, height, fetcher, loadProvider,
transformation,
                transcoder, diskCacheProvider,
diskCacheStrategy, priority);
        EngineRunnable runnable = new
EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load",
startTime, key);
```

```
        }

        return new LoadStatus(cb, engineJob);
    }

    ...
}
```

可以看到，这里在第17行调用了loadFromCache()方法来获取缓存图片，如果获取到就直接调用cb.onResourceReady()方法进行回调。如果没有获取到，则会在第26行调用loadFromActiveResources()方法来获取缓存图片，获取到的话也直接进行回调。只有在两个方法都没有获取到缓存的情况下，才会继续向下执行，从而开启线程来加载图片。

也就是说，Glide的图片加载过程中会调用两个方法来获取内存缓存，loadFromCache()和loadFromActiveResources()。这两个方法中一个使用的就是LruCache算法，另一个使用的就是弱引用。我们来看一下它们的源码：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    private final MemoryCache cache;
    private final Map<Key, WeakReference<EngineResource<?>>> activeResources;
    ...

    private EngineResource<?> loadFromCache(Key key,
        boolean isMemoryCacheable) {
        if (!isMemoryCacheable) {
            return null;
        }
        EngineResource<?> cached =
        getEngineResourceFromCache(key);
        if (cached != null) {
            cached.acquire();
            activeResources.put(key, new
                ResourceWeakReference(key, cached, getReferenceQueue())));
        }
    }
}
```

```
        }

        return cached;
    }

    private EngineResource<?>
getEngineResourceFromCache(Key key) {
    Resource<?> cached = cache.remove(key);
    final EngineResource result;
    if (cached == null) {
        result = null;
    } else if (cached instanceof EngineResource) {
        result = (EngineResource) cached;
    } else {
        result = new EngineResource(cached, true
/*isCacheable*/);
    }
    return result;
}

private EngineResource<?> loadFromActiveResources(Key
key, boolean isMemoryCacheable) {
    if (!isMemoryCacheable) {
        return null;
    }
    EngineResource<?> active = null;
    WeakReference<EngineResource<?>> activeRef =
activeResources.get(key);
    if (activeRef != null) {
        active = activeRef.get();
        if (active != null) {
            active.acquire();
        } else {
            activeResources.remove(key);
        }
    }
    return active;
}
```

```
    ...  
}
```

在loadFromCache()方法的一开始，首先就判断了isMemoryCacheable是不是false，如果是false的话就直接返回null。这是什么意思呢？其实很简单，我们刚刚不是学了一个skipMemoryCache()方法吗？如果在这个方法中传入true，那么这里的isMemoryCacheable就会是false，表示内存缓存已被禁用。

我们继续往下看，接着调用了getEngineResourceFromCache()方法来获取缓存。在这个方法中，会使用缓存Key来从cache当中取值，而这里的cache对象就是在构建Glide对象时创建的LruResourceCache，那么说明这里其实使用的就是LruCache算法了。

但是呢，观察第22行，当我们从LruResourceCache中获取到缓存图片之后会将它从缓存中移除，然后在第16行将这个缓存图片存储到activeResources当中。activeResources就是一个弱引用的HashMap，用来缓存正在使用中的图片，我们可以看到，loadFromActiveResources()方法就是从activeResources这个HashMap当中取值的。使用activeResources来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

好的，从内存缓存中读取数据的逻辑大概就是这些了。概括一下来说，就是如果能从内存缓存当中读取到要加载的图片，那么就直接进行回调，如果读取不到的话，才会开启线程执行后面的图片加载逻辑。

现在我们已经搞明白了内存缓存读取的原理，接下来的问题就是内存缓存是在哪里写入的呢？这里我们又要回顾一下上一篇文章中的内容了。还记得我们之前分析过，当图片加载完成之后，会在EngineJob当中通过Handler发送一条消息将执行逻辑切回到主线程当中，从而执行handleResultOnMainThread()方法。那么我们现在重新来看一下这个方法，代码如下所示：

```
class EngineJob implements  
EngineRunnable.EngineRunnableManager {
```

```
    private final EngineResourceFactory
engineResourceFactory;
    ...

    private void handleResultOnMainThread() {
        if (isCancelled) {
            resource.recycle();
            return;
        } else if (cbs.isEmpty()) {
            throw new IllegalStateException("Received a
resource without any callbacks to notify");
        }
        engineResource =
engineResourceFactory.build(resource, isCacheable);
        hasResource = true;
        engineResource.acquire();
        listener.onEngineJobComplete(key,
engineResource);
        for (ResourceCallback cb : cbs) {
            if (!isInIgnoredCallbacks(cb)) {
                engineResource.acquire();
                cb.onResourceReady(engineResource);
            }
        }
        engineResource.release();
    }

    static class EngineResourceFactory {
        public <R> EngineResource<R> build(Resource<R>
resource, boolean isMemoryCacheable) {
            return new EngineResource<R>(resource,
isMemoryCacheable);
        }
    }
    ...
}
```

在第13行，这里通过EngineResourceFactory构建出了一个包含图片资源的EngineResource对象，然后会在第16行将这个对象回调到Engine的onEngineJobComplete()方法当中，如下所示：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {
    ...
    @Override
    public void onEngineJobComplete(Key key,
        EngineResource<?> resource) {
        Util.assertMainThread();
        // A null resource indicates that the load
        failed, usually due to an exception.
        if (resource != null) {
            resource.setResourceListener(key, this);
            if (resource.isCacheable()) {
                activeResources.put(key, new
                    ResourceWeakReference(key, resource,
                        getReferenceQueue()));
            }
        }
        jobs.remove(key);
    }
    ...
}
```

现在就非常明显了，可以看到，在第13行，回调过来的EngineResource被put到了activeResources当中，也就是在这里写入的缓存。

那么这只是弱引用缓存，还有另外一种LruCache缓存是在哪里写入的呢？这就要介绍一下EngineResource中的一个引用机制了。观察刚才的handleResultOnMainThread()方法，在第15行和第19行有调用EngineResource的acquire()方法，在第23行有调用它的release()方法。其实，EngineResource是用一个acquired变量用来记录图片被引用的次

数，调用acquire()方法会让变量加1，调用release()方法会让变量减1，代码如下所示：

```
class EngineResource<Z> implements Resource<Z> {

    private int acquired;
    ...

    void acquire() {
        if (isRecycled) {
            throw new IllegalStateException("Cannot
acquire a recycled resource");
        }
        if
(!Looper.getMainLooper().equals(Looper.myLooper())) {
            throw new IllegalThreadStateException("Must
call acquire on the main thread");
        }
        ++acquired;
    }

    void release() {
        if (acquired <= 0) {
            throw new IllegalStateException("Cannot
release a recycled or not yet acquired resource");
        }
        if
(!Looper.getMainLooper().equals(Looper.myLooper())) {
            throw new IllegalThreadStateException("Must
call release on the main thread");
        }
        if (--acquired == 0) {
            listener.onResourceReleased(key, this);
        }
    }
}
```

也就是说，当acquired变量大于0的时候，说明图片正在使用中，也就应该放到activeResources弱引用缓存当中。而经过release()之后，如果acquired变量等于0了，说明图片已经不再被使用了，那么此时会在第24行调用listener的onResourceReleased()方法来释放资源，这个listener就是Engine对象，我们来看下它的onResourceReleased()方法：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    private final MemoryCache cache;
    private final Map<Key, WeakReference<EngineResource>?>>> activeResources;
    ...

    @Override
    public void onResourceReleased(Key cacheKey,
        EngineResource resource) {
        Util.assertMainThread();
        activeResources.remove(cacheKey);
        if (resource.isCacheable()) {
            cache.put(cacheKey, resource);
        } else {
            resourceRecycler.recycle(resource);
        }
    }

    ...
}
```

可以看到，这里首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

这就是Glide内存缓存的实现原理。

总结：

内存缓存读取：先获取LruResourceCache（Lru算法缓存），然后放入activeResources（是一个弱引用的HashMap）来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

内存缓存写入：回调过来的EngineResource先被put到了activeResources当中，也就是在这里写入的缓存。如果图片正在使用中，也就应该放到activeResources弱引用缓存当中。如果图片不再被使用了，首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

进一步总结：正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

磁盘缓存：

接下来我们开始学习硬盘缓存方面的内容。

主要作用就是防止重复将图片读取到我们的磁盘当中

磁盘缓存读取：

不知道你还记不记得，在本系列的第一篇文章中我们就使用过硬盘缓存的功能了。当时为了禁止Glide对图片进行硬盘缓存而使用了如下代码：

```
glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

调用diskCacheStrategy()方法并传入DiskCacheStrategy.NONE，就可以禁用掉Glide的硬盘缓存功能了。

这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收四种参数：

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.SOURCE： 表示只缓存原始图片。

- DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。
- DiskCacheStrategy.ALL : 表示既缓存原始图片，也缓存转换过后的图片。

上面四种参数的解释本身并没有什么难理解的地方，但是有一个概念大家需要了解，就是当我们使用Glide去加载一张图片的时候，Glide默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在后面学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。而Glide默认情况下在硬盘缓存的就是转换过后的图片，我们通过调用diskCacheStrategy()方法则可以改变这一默认行为。

好的，关于Glide硬盘缓存的用法也就只有这么多，那么接下来还是老套路，我们通过阅读源码来分析一下，Glide的硬盘缓存功能是如何实现的。

首先，和内存缓存类似，硬盘缓存的实现也是使用的LruCache算法，而且Google还提供了一个现成的工具类DiskLruCache。我之前也专门写过一篇文章对这个DiskLruCache工具进行了比较全面的分析，感兴趣的朋友可以参考一下 Android DiskLruCache完全解析，硬盘缓存的最佳方案。当然，Glide是使用的自己编写的DiskLruCache工具类，但是基本的实现原理都是差不多的。

接下来我们看一下Glide是在哪里读取硬盘缓存的。这里又需要回忆一下上篇文章中的内容了，Glide开启线程来加载图片后会执行 EngineRunnable的run()方法，run()方法中又会调用一个decode()方法，那么我们重新再来看一下这个decode()方法的源码：

```
private Resource<?> decode() throws Exception {
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}
```

可以看到，这里会分为两种情况，一种是调用decodeFromCache()方法从硬盘缓存当中读取图片，一种是调用decodeFromSource()来读取原始图片。默认情况下Glide会优先从缓存当中读取，只有缓存中不存在要读取的图片时，才会去读取原始图片。那么我们现在来看一下decodeFromCache()方法的源码，如下所示：

```
private Resource<?> decodeFromCache() throws Exception {  
    Resource<?> result = null;  
    try {  
        result = decodeJob.decodeResultFromCache();  
    } catch (Exception e) {  
        if (Log.isLoggable(TAG, Log.DEBUG)) {  
            Log.d(TAG, "Exception decoding result from  
cache: " + e);  
        }  
    }  
    if (result == null) {  
        result = decodeJob.decodeSourceFromCache();  
    }  
    return result;  
}
```

可以看到，这里会先去调用DecodeJob的decodeResultFromCache()方法来获取缓存，如果获取不到，会再调用decodeSourceFromCache()方法获取缓存，这两个方法的区别其实就是DiskCacheStrategy.RESULT和DiskCacheStrategy.SOURCE这两个参数的区别，相信不需要我再做什么解释吧。

那么我们来看一下这两个方法的源码吧，如下所示：

```
public Resource<Z> decodeResultFromCache() throws  
Exception {  
    if (!diskCacheStrategy.cacheResult()) {  
        return null;  
    }  
    long startTime = LogTime.getLogTime();  
    Resource<T> transformed = loadFromCache(resultKey);
```

```
startTime = LogTime.getLogTime();
Resource<Z> result = transcode(transformed);
return result;
}

public Resource<Z> decodeSourceFromCache() throws
Exception {
    if (!diskCacheStrategy.cacheSource()) {
        return null;
    }
    long startTime = LogTime.getLogTime();
    Resource<T> decoded =
loadFromCache(resultKey.getOriginalKey());
    return transformEncodeAndTranscode(decoded);
}
```

可以看到，它们都是调用了loadFromCache()方法从缓存当中读取数据，如果是decodeResultFromCache()方法就直接将数据解码并返回，如果是decodeSourceFromCache()方法，还要调用一下transformEncodeAndTranscode()方法先将数据转换一下再解码并返回。

然而我们注意到，这两个方法中在调用loadFromCache()方法时传入的参数却不一样，一个传入的是resultKey，另外一个却又调用了resultKey的getOriginalKey()方法。这个其实非常好理解，刚才我们已经解释过了，Glide的缓存Key是由10个参数共同组成的，包括图片的width、height等等。但如果我们要缓存的原始图片，其实并不需要这么多的参数，因为不用对图片做任何的变化。那么我们来看一下getOriginalKey()方法的源码：

```
public Key getOriginalKey() {
    if (originalKey == null) {
        originalKey = new OriginalKey(id, signature);
    }
    return originalKey;
}
```

可以看到，这里其实就是忽略了绝大部分的参数，只使用了id和signature这两个参数来构成缓存Key。而signature参数绝大多数情况下都是用不到的，因此基本上可以说就是由id（也就是图片url）来决定的Original缓存Key。

搞明白了这两种缓存Key的区别，那么接下来我们看一下loadFromCache()方法的源码吧：

```
private Resource<T> loadFromCache(Key key) throws  
IOException {  
    File cacheFile =  
    diskCacheProvider.getDiskCache().get(key);  
    if (cacheFile == null) {  
        return null;  
    }  
    Resource<T> result = null;  
    try {  
        result =  
    loadProvider.getCacheDecoder().decode(cacheFile, width,  
height);  
    } finally {  
        if (result == null) {  
            diskCacheProvider.getDiskCache().delete(key);  
        }  
    }  
    return result;  
}
```

这个方法的逻辑非常简单，调用getDiskCache()方法获取到的就是Glide自己编写的DiskLruCache工具类的实例，然后调用它的get()方法并把缓存Key传入，就能得到硬盘缓存的文件了。如果文件为空就返回null，如果文件不为空则将它解码成Resource对象后返回即可。

这样我们就将硬盘缓存读取的源码分析完了，那么硬盘缓存又是在哪里写入的呢？趁热打铁我们赶快继续分析下去。

磁盘缓存写入：

刚才已经分析过了，在没有缓存的情况下，会调用decodeFromSource()方法来读取原始图片。那么我们来看下这个方法：

```
public Resource<Z> decodeFromSource() throws Exception {  
    Resource<T> decoded = decodeSource();  
    return transformEncodeAndTranscode(decoded);  
}
```

这个方法中只有两行代码，decodeSource()顾名思义是用来解析原图片的，而transformEncodeAndTranscode()则是用来对图片进行转换和转码的。我们先来看decodeSource()方法：

```
private Resource<T> decodeSource() throws Exception {  
    Resource<T> decoded = null;  
    try {  
        long startTime = LogTime.getLogTime();  
        final A data = fetcher.loadData(priority);  
        if (isCancelled) {  
            return null;  
        }  
        decoded = decodeFromSourceData(data);  
    } finally {  
        fetcher.cleanup();  
    }  
    return decoded;  
}  
  
private Resource<T> decodeFromSourceData(A data) throws  
IOException {  
    final Resource<T> decoded;  
    if (diskCacheStrategy.cacheSource()) {  
        decoded = cacheAndDecodeSourceData(data);  
    } else {  
        long startTime = LogTime.getLogTime();  
        decoded =  
loadProvider.getSourceDecoder().decode(data, width,  
height);  
    }  
}
```

```
        return decoded;
    }

private Resource<T> cacheAndDecodeSourceData(A data)
throws IOException {
    long startTime = LogTime.getLogTime();
    SourceWriter<A> writer = new SourceWriter<A>
    (loadProvider.getSourceEncoder(), data);

    diskCacheProvider.getDiskCache().put(resultKey.getOriginalKey(), writer);
    startTime = LogTime.getLogTime();
    Resource<T> result =
    loadFromCache(resultKey.getOriginalKey());
    return result;
}
```

这里会在第5行先调用fetcher的loadData()方法读取图片数据，然后在第9行调用decodeFromSourceData()方法来对图片进行解码。接下来会在第18行先判断是否允许缓存原始图片，如果允许的话又会调用cacheAndDecodeSourceData()方法。而在这个方法中同样调用了getDiskCache()方法来获取DiskLruCache实例，接着调用它的put()方法就可以写入硬盘缓存了，注意原始图片的缓存Key是用的resultKey.getOriginalKey()。

好的，原始图片的缓存写入就是这么简单，接下来我们分析一下transformEncodeAndTranscode()方法的源码，来看看转换过后的图片缓存是怎么写入的。代码如下所示：

```
private Resource<Z>
transformEncodeAndTranscode(Resource<T> decoded) {
    long startTime = LogTime.getLogTime();
    Resource<T> transformed = transform(decoded);
    writeTransformedToCache(transformed);
    startTime = LogTime.getLogTime();
    Resource<Z> result = transcode(transformed);
    return result;
}
```

```
private void writeTransformedToCache(Resource<T> transformed) {
    if (transformed == null || !diskCacheStrategy.cacheResult()) {
        return;
    }
    long startTime = LogTime.getLogTime();
    SourceWriter<Resource<T>> writer = new
    SourceWriter<Resource<T>>(loadProvider.getEncoder(),
    transformed);
    diskCacheProvider.getDiskCache().put(resultKey,
    writer);
}
```

这里的逻辑就更加简单明了了。先是在第3行调用transform()方法来对图片进行转换，然后在writeTransformedToCache()方法中将转换过后的图片写入到硬盘缓存中，调用的同样是DiskLruCache实例的put()方法，不过这里用的缓存Key是resultKey。

这样我们就将Glide硬盘缓存的实现原理也分析完了。虽然这些源码看上去如此的复杂，但是经过Glide出色的封装，使得我们只需要通过skipMemoryCache()和diskCacheStrategy()这两个方法就可以轻松自如地控制Glide的缓存功能了。

了解了Glide缓存的实现原理之后，接下来我们再来学习一些Glide缓存的高级技巧吧。

高级技巧

虽说Glide将缓存功能高度封装之后，使得用法变得非常简单，但同时也带来了一些问题。

比如之前有一位群里的朋友就跟我说过，他们项目的图片资源都是存放在七牛云上面的，而七牛云为了对图片资源进行保护，会在图片url地址的基础之上再加上一个token参数。也就是说，一张图片的url地址可能会是如下格式：

```
http://url.com/image.jpg?token=d9caa6e02c990b0a
```

而使用Glide加载这张图片的话，也就会使用这个url地址来组成缓存Key。

但是接下来问题就来了，token作为一个验证身份的参数并不是一成不变的，很有可能时时刻刻都在变化。而如果token变了，那么图片的url也就跟着变了，图片url变了，缓存Key也就跟着变了。结果就造成了，明明是同一张图片，就因为token不断在改变，导致Glide的缓存功能完全失效了。

这其实是个挺棘手的问题，而且我相信绝对不仅仅是七牛云这一个个例，大家在使用Glide的时候很有可能都会遇到这个问题。

那么该如何解决这个问题呢？我们还是从源码的层面进行分析，首先再来看一下Glide生成缓存Key这部分的代码：

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {  
  
    public <T, Z, R> LoadStatus load(Key signature, int  
width, int height, DataFetcher<T> fetcher,  
        DataLoadProvider<T, Z> loadProvider,  
        Transformation<Z> transformation, ResourceTranscoder<Z,  
R> transcoder,  
        Priority priority, boolean isMemoryCacheable,  
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)  
{  
    util.assertMainThread();  
    long startTime = LogTime.getLogTime();  
  
    final String id = fetcher.getId();  
    EngineKey key = keyFactory.buildKey(id,  
signature, width, height, loadProvider.getCacheDecoder(),  
        loadProvider.getSourceDecoder(),  
        transformation, loadProvider.getEncoder(),  
        transcoder,  
    loadProvider.getSourceEncoder());
```

```
    ...  
}
```

```
    ...  
}
```

来看一下第11行，刚才已经说过了，这个id其实就是图片的url地址。那么，这里是通过调用fetcher.getId()方法来获取的图片url地址，而我们在上一篇文章中已经知道了，fetcher就是HttpUrlFetcher的实例，我们就来看一下它的getId()方法的源码吧，如下所示：

```
public class HttpUrlFetcher implements  
DataFetcher<InputStream> {  
  
    private final GlideUrl glideUrl;  
    ...  
  
    public HttpUrlFetcher(GlideUrl glideUrl) {  
        this(glideUrl, DEFAULT_CONNECTION_FACTORY);  
    }  
  
    public HttpUrlFetcher(GlideUrl glideUrl,  
HttpURLConnectionFactory connectionFactory) {  
        this.glideUrl = glideUrl;  
        this.connectionFactory = connectionFactory;  
    }  
  
    @Override  
    public String getId() {  
        return glideUrl.getCacheKey();  
    }  
  
    ...  
}
```

可以看到，`getId()`方法中又调用了`GlideUrl`的`getCacheKey()`方法。那么这个`GlideUrl`对象是从哪里来的呢？其实就是在`load()`方法中传入的图片url地址，然后Glide在内部把这个url地址包装成了一个`GlideUrl`对象。

很明显，接下来我们就要看一下`GlideUrl`的`getCacheKey()`方法的源码了，如下所示：

```
public class GlideUrl {  
  
    private final URL url;  
    private final String stringUrl;  
    ...  
  
    public GlideUrl(URL url) {  
        this(url, Headers.DEFAULT);  
    }  
  
    public GlideUrl(String url) {  
        this(url, Headers.DEFAULT);  
    }  
  
    public GlideUrl(URL url, Headers headers) {  
        ...  
        this.url = url;  
        stringUrl = null;  
    }  
  
    public GlideUrl(String url, Headers headers) {  
        ...  
        this.stringUrl = url;  
        this.url = null;  
    }  
  
    public String getCacheKey() {  
        return stringUrl != null ? stringUrl :  
url.toString();  
    }  
}
```

```
    ...  
}
```

这里我将代码稍微进行了一点简化，这样看上去更加简单明了。GlideUrl类的构造函数接收两种类型的参数，一种是url字符串，一种是URL对象。然后getCacheKey()方法中的判断逻辑非常简单，如果传入的是url字符串，那么就直接返回这个字符串本身，如果传入的是URL对象，那么就返回这个对象toString()后的结果。

其实看到这里，我相信大家已经猜到解决方案了，因为getCacheKey()方法中的逻辑太直白了，直接就是将图片的url地址进行返回来作为缓存Key的。那么其实我们只需要重写这个getCacheKey()方法，加入一些自己的逻辑判断，就能轻松解决掉刚才的问题了。

创建一个MyGlideUrl继承自GlideUrl，代码如下所示：

```
public class MyGlideUrl extends GlideUrl {  
  
    private String mUrl;  
  
    public MyGlideUrl(String url) {  
        super(url);  
        mUrl = url;  
    }  
  
    @Override  
    public String getCacheKey() {  
        return mUrl.replace(findTokenParam(), "");  
    }  
  
    private String findTokenParam() {  
        String tokenParam = "";  
        int tokenKeyIndex = mUrl.indexOf("?token=") >= 0  
            ? mUrl.indexOf("?token=") : mUrl.indexOf("&token=");  
        if (tokenKeyIndex != -1) {
```

```
        int nextAndIndex = mUrl.indexOf("&",  
tokenKeyIndex + 1);  
        if (nextAndIndex != -1) {  
            tokenParam = mUrl.substring(tokenKeyIndex  
+ 1, nextAndIndex + 1);  
        } else {  
            tokenParam =  
mUrl.substring(tokenKeyIndex);  
        }  
    }  
  
    return tokenParam;  
}  
  
}
```

可以看到，这里我们重写了getCacheKey()方法，在里面加入了一段逻辑用于将图片url地址中token参数的这一部分移除掉。这样getCacheKey()方法得到的就是一个没有token参数的url地址，从而不管token怎么变化，最终Glide的缓存Key都是固定不变的了。

当然，定义好了MyGlideUrl，我们还得使用它才行，将加载图片的代码改成如下方式即可：

```
Glide.with(this)  
.load(new MyGlideUrl(url))  
.into(imageview);
```

也就是说，我们需要在load()方法中传入这个自定义的MyGlideUrl对象，而不能再像之前那样直接传入url字符串了。不然的话Glide在内部还是会使用原始的GlideUrl类，而不是我们自定义的MyGlideUrl类。

这样我们就将这个棘手的缓存问题给解决掉了。

3.5 Glide源码学习五：回调与监听

回调的源码实现

作为一名Glide老手，相信大家对于Glide的基本用法已经非常熟练了。我们都清楚，使用Glide在界面上加载并展示一张图片只需要一行代码：

```
Glide.with(this).load(url).into(imageview);
```

而在这一行代码的背后，Glide帮我们执行了成千上万行的逻辑。其实在第二篇文章当中，我们已经分析了这一行代码背后的完整执行流程，但是这里我准备再带着大家单独回顾一下回调这部分的源码，这将有助于我们今天这篇文章的学习。

首先来看一下into()方法，这里我们将ImageView的实例传入到into()方法当中，Glide将图片加载完成之后，图片就能显示到ImageView上了。这是怎么实现的呢？我们来看一下into()方法的源码：

```
public Target<TranscodeType> into(ImageView view) {  
    Util.assertMainThread();  
    if (view == null) {  
        throw new IllegalArgumentException("You must pass  
in a non null view");  
    }  
    if (!isTransformationSet && view.getScaleType() !=  
null) {  
        switch (view.getScaleType()) {  
            case CENTER_CROP:  
                applyCenterCrop();  
                break;  
            case FIT_CENTER:  
            case FIT_START:  
            case FIT_END:  
                applyFitCenter();  
                break;  
            default:  
                // Do nothing.  
        }  
    }  
    return into(glide.buildImageViewTarget(view,  
transcodeClass));  
}
```

可以看到，最后一行代码会调用glide.buildImageViewTarget()方法构建出一个Target对象，然后再把它传入到另一个接收Target参数的into()方法中。Target对象则是用来最终展示图片用的，如果我们跟进到glide.buildImageViewTarget()方法中，你会看到如下的源码：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view,
Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz))
{
            return (Target<Z>) new
GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new
BitmapImageViewTarget(view);
        } else if
(Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new
DrawableImageViewTarget(view);
        } else {
            throw new IllegalArgumentException("Unhandled
class: " + clazz
                    + ", try .as*
(Class).transcode(ResourceTranscoder)");
        }
    }
}
```

buildTarget()方法会根据传入的class参数来构建不同的Target对象，如果你在使用Glide加载图片的时候调用了asBitmap()方法，那么这里就会构建出BitmapImageViewTarget对象，否则的话构建的都是GlideDrawableImageViewTarget对象。至于上述代码中的DrawableImageViewTarget对象，这个通常都是用不到的，我们可以暂时不用管它。

之后就会把这里构建出来的Target对象传入到GenericRequest当中，而Glide在图片加载完成之后又会回调GenericRequest的onResourceReady()方法，我们来看一下这部分源码：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback, ResourceCallback {

    private Target<R> target;
    ...

    private void onResourceReady(Resource<?> resource, R result) {
        boolean isFirstResource = isFirstReadyResource();
        status = Status.COMPLETE;
        this.resource = resource;
        if (requestListener == null ||
            !requestListener.onResourceReady(result, model, target,
                loadedFromMemoryCache, isFirstResource))
        {
            GlideAnimation<R> animation =
            animationFactory.build(loadedFromMemoryCache,
            isFirstResource);
            target.onResourceReady(result, animation);
        }
        notifyLoadSuccess();
    }
    ...
}
```

这里在第14行调用了target.onResourceReady()方法，而刚才我们已经知道，这里的target就是GlideDrawableImageViewTarget对象，那么我们再来看一下它的源码：

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideDrawable> {

    ...
}
```

```
@Override
public void onResourceReady(GlideDrawable resource,
GlideAnimation<? super GlideDrawable> animation) {
    if (!resource.isAnimated()) {
        float viewRatio = view.getWidth() / (float)
view.getHeight();
        float drawableRatio =
resource.getIntrinsicWidth() / (float)
resource.getIntrinsicHeight();
        if (Math.abs(viewRatio - 1f) <=
SQUARE_RATIO_MARGIN
            && Math.abs(drawableRatio - 1f) <=
SQUARE_RATIO_MARGIN) {
            resource = new SquaringDrawable(resource,
view.getWidth());
        }
    }
    super.onResourceReady(resource, animation);
    this.resource = resource;
    resource.setLoopCount(maxLoopCount);
    resource.start();
}

@Override
protected void setResource(GlideDrawable resource) {
    view.setImageDrawable(resource);
}

...
}
```

可以看到，这里在onResourceReady()方法中处理了图片展示，还有GIF播放的逻辑，那么一张图片也就显示出来了，这也就是Glide回调的基本实现原理。

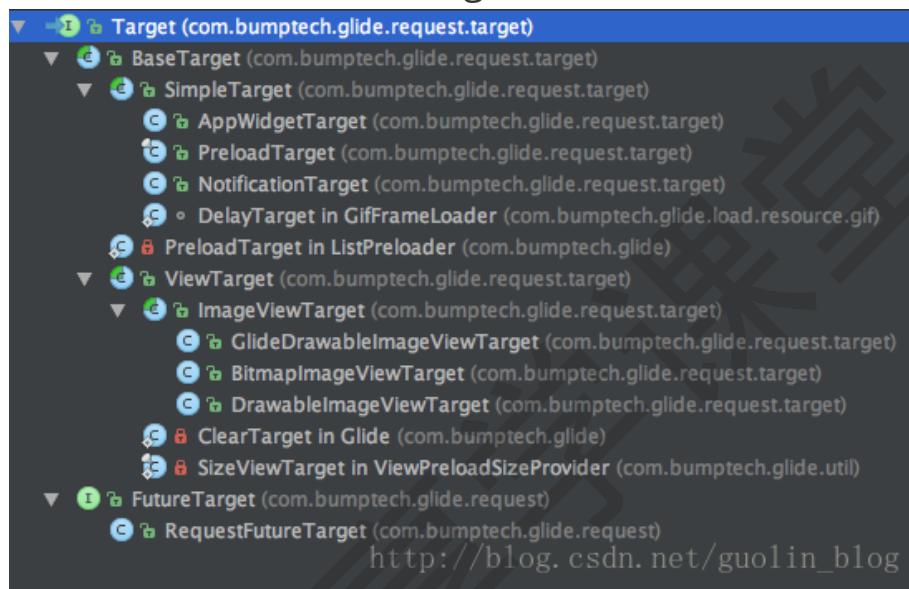
好的，那么原理就先分析到这儿，接下来我们就来看一下在回调和监听方面还有哪些知识是可以扩展的。

into()方法

使用了这么久的Glide，我们都知道into()方法中是可以传入ImageView的。那么into()方法还可以传入别的参数吗？我可以让Glide加载出来的图片不显示到ImageView上吗？答案是肯定的，这就需要用到自定义Target功能。

其实通过上面的分析，我们已经知道了，into()方法还有一个接收Target参数的重载。即使我们传入的参数是ImageView，Glide也会在内部自动构建一个Target对象。而如果我们能够掌握自定义Target技术的话，就可以更加随心所欲地控制Glide的回调了。

我们先来看一下Glide中Target的继承结构图吧，如下所示：



可以看到，Target的继承结构还是相当复杂的，实现Target接口的子类非常多。不过你不用被这么多的子类所吓到，这些大多数都是Glide已经实现好的具备完整功能的Target子类，如果我们要进行自定义的话，通常只需要在两种Target的基础上去自定义就可以了，一种是SimpleTarget，一种是ViewTarget。

接下来我就分别以这两种Target来举例，学习一下自定义Target的功能。

首先来看SimpleTarget，顾名思义，它是一种极为简单的Target，我们使用它可以将Glide加载出来的图片对象获取到，而不是像之前那样只能将图片在ImageView上显示出来。

那么下面我们来看一下SimpleTarget的用法示例吧，其实非常简单：

```
simpleTarget<GlideDrawable> simpleTarget = new
simpleTarget<GlideDrawable>() {
    @Override
    public void onResourceReady(GlideDrawable resource,
GlideAnimation glideAnimation) {
        imageView.setImageDrawable(resource);
    }
};

public void loadImage(View view) {
    String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
    Glide.with(this)
        .load(url)
        .into(simpleTarget);
}
```

怎么样？不愧是SimpleTarget吧，短短几行代码就搞了。这里我们创建了一个SimpleTarget的实例，并且指定它的泛型是GlideDrawable，然后重写了onResourceReady()方法。在onResourceReady()方法中，我们就可以获取到Glide加载出来的图片对象了，也就是方法参数中传过来的GlideDrawable对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了ImageView上。

SimpleTarget的实现创建好了，那么只需要在加载图片的时候将它传入到into()方法中就可以了，现在运行一下程序，效果如下图所示。



虽然目前这个效果和直接在into()方法中传入ImageView并没有什么区别，但是我们已经拿到了图片对象的实例，然后就可以随意做更多的事情了。

当然，SimpleTarget中的泛型并不一定只能是GlideDrawable，如果你能确定你正在加载的是一张静态图而不是GIF图的话，我们还能直接拿到这张图的Bitmap对象，如下所示：

```
SimpleTarget<Bitmap> simpleTarget = new
SimpleTarget<Bitmap>() {
    @Override
    public void onResourceReady(Bitmap resource,
        GlideAnimation glideAnimation) {
        imageView.setImageBitmap(resource);
    }
};

public void loadImage(View view) {
    String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
    Glide.with(this)
        .load(url)
```

```
.asBitmap()
.into(simpleTarget);
}
```

可以看到，这里我们将SimpleTarget的泛型指定成Bitmap，然后在加载图片的时候调用了asBitmap()方法强制指定这是一张静态图，这样就能在onResourceReady()方法中获取到这张图的Bitmap对象了。

好了，SimpleTarget的用法就是这么简单，接下来我们学习一下ViewTarget的用法。

事实上，从刚才的继承结构图上就能看出，Glide在内部自动帮我们创建的GlideDrawableImageViewTarget就是ViewTarget的子类。只不过GlideDrawableImageViewTarget被限定只能作用在ImageView上，而ViewTarget的功能更加广泛，它可以作用在任意的View上。

这里我们还是通过一个例子来演示一下吧，比如我创建了一个自定义布局MyLayout，如下所示：

```
public class MyLayout extends LinearLayout {

    private ViewTarget<MyLayout, GlideDrawable>
viewTarget;

    public MyLayout(Context context, AttributeSet attrs)
{
    super(context, attrs);
    viewTarget = new ViewTarget<MyLayout,
GlideDrawable>(this) {
        @Override
        public void onResourceReady(GlideDrawable
resource, GlideAnimation glideAnimation) {
            MyLayout myLayout = getView();
            myLayout.setImageResource(resource);
        }
    };
}
```

```
public ViewTarget<MyLayout, GlideDrawable>
getTarget() {
    return viewTarget;
}

public void setImageAsBackground(GlideDrawable
resource) {
    setBackground(resource);
}

}
```

在MyLayout的构造函数中，我们创建了一个ViewTarget的实例，并将MyLayout当前的实例this传了进去。ViewTarget中需要指定两个泛型，一个是View的类型，一个图片的类型（GlideDrawable或Bitmap）。然后在onResourceReady()方法中，我们就可以通过getView()方法获取到MyLayout的实例，并调用它的任意接口了。比如说这里我们调用了setImageAsBackground()方法来将加载出来的图片作为MyLayout布局的背景图。

接下来看一下怎么使用这个Target吧，由于MyLayout中已经提供了getTarget()接口，我们只需要在加载图片的地方这样写就可以了：

```
public class MainActivity extends AppCompatActivity {

    MyLayout myLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myLayout = (MyLayout)
        findViewById(R.id.background);
    }

    public void loadImage(View view) {
```

```
        String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
        Glide.with(this)
            .load(url)
            .into(myLayout.getTarget());
    }

}
```

就是这么简单，在into()方法中传入myLayout.getTarget()即可。现在重新运行一下程序，效果如下图所示。



好的，关于自定义Target的功能我们就介绍这么多，这些虽说都是自定义Target最基本的用法，但掌握了这些用法之后，你就能应对各种各样复杂的逻辑了。

preload()方法

Glide加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待漫长的网络加载时间了，这该怎么办呢？

对于很多Glide新手来说这确实是一个烦恼的问题，因为在没有学习本篇文章之前，into()方法中必须传入一个ImageView呀，而传了ImageView之后图片就显示出来了，这还怎么预加载呢？

不过在学习了本篇文章之后，相信你已经能够想到解决方案了。因为into()方法中除了传入ImageView之后还可以传入Target对象，如果我们在Target对象的onResourceReady()方法中做一个空实现，也就是不做任何逻辑处理，那么图片自然也就显示不出来了，而Glide的缓存机制却仍然还会正常工作，这样不就实现预加载功能了吗？

没错，上述的做法完全可以实现预加载功能，不过有没有感觉这种实现方式有点笨笨的。事实上，Glide专门给我们提供了预加载的接口，也就是preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换into()方法即可，如下所示：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .preload();
```

需要注意的是，我们如果使用了preload()方法，最好要将diskCacheStrategy的缓存策略指定成DiskCacheStrategy.SOURCE。因为preload()方法默认是预加载的原始图片大小，而into()方法则默认会根据ImageView控件的大小来动态决定加载图片的大小。因此，如果不将diskCacheStrategy的缓存策略指定成DiskCacheStrategy.SOURCE的话，很容易会造成我们在预加载完成之后再使用into()方法加载图片，却仍然还是要从网络上去请求图片这种现象。

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为Glide会直接从缓存当中去读取图片并显示出来，代码如下所示：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .into(imageview);
```

注意，这里我们仍然需要使用diskCacheStrategy()方法将硬盘缓存策略指定成DiskCacheStrategy.SOURCE，以保证Glide一定会去读取刚才预加载的图片缓存。

preload()方法的用法大概就是这么简单，但是仅仅会使用显然层次有些太低了，下面我们就满足一下好奇心，看看它的源码是如何实现的。

和into()方法一样，preload()方法也是在GenericRequestBuilder类当中的，代码如下所示：

```
public class GenericRequestBuilder<ModelType, DataType,
ResourceType, TranscodeType> implements Cloneable {
    ...
    public Target<TranscodeType> preload(int width, int
height) {
        final PreloadTarget<TranscodeType> target =
PreloadTarget.obtain(width, height);
        return into(target);
    }

    public Target<TranscodeType> preload() {
        return preload(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL);
    }

    ...
}
```

正如刚才所说，preload()方法有两个方法重载，你可以调用带参数的preload()方法来明确指定图片的宽和高，也可以调用不带参数的preload()方法，它会在内部自动将图片的宽和高都指定成Target.SIZE_ORIGINAL，也就是图片的原始尺寸。

然后我们可以看到，这里在第5行调用了PreloadTarget.obtain()方法获取一个PreloadTarget的实例，并把它传入到了into()方法当中。从刚才的继承结构图中可以看出，PreloadTarget是SimpleTarget的子类，因此它是可以直接传入到into()方法中的。

那么现在的问题就是，PreloadTarget具体的实现到底是什么样子的了，我们看一下它的源码，如下所示：

```
public final class PreloadTarget<Z> extends  
SimpleTarget<Z> {  
  
    public static <Z> PreloadTarget<Z> obtain(int width,  
    int height) {  
        return new PreloadTarget<Z>(width, height);  
    }  
  
    private PreloadTarget(int width, int height) {  
        super(width, height);  
    }  
  
    @Override  
    public void onResourceReady(Z resource,  
    GlideAnimation<? super Z> glideAnimation) {  
        Glide.clear(this);  
    }  
}
```

PreloadTarget的源码非常简单，obtain()方法中就是new了一个PreloadTarget的实例而已，而onResourceReady()方法中也没做什么事情，只是调用了Glide.clear()方法。

这里的Glide.clear()并不是清空缓存的意思，而是表示加载已完成，释放资源的意思，因此不用在这里产生疑惑。

其实PreloadTarget的思想和我们刚才提到设计思路是一样的，就是什么都不做就可以了。因为图片加载完成之后只将它缓存而不去显示它，那不就相当于预加载了嘛。

preload()方法不管是在用法方面还是源码实现方面都还是非常简单的，那么关于这个方法我们就学到这里。

downloadOnly()方法

一直以来，我们使用Glide都是为了将图片显示到界面上。虽然我们知道Glide会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实Glide将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到downloadOnly()方法了。

和preload()方法类似，downloadOnly()方法也是可以替换into()方法的，不过downloadOnly()方法的用法明显要比preload()方法复杂不少。顾名思义，downloadOnly()方法表示只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。downloadOnly()方法是定义在DrawableTypeRequest类当中的，它有两个方法重载，一个接收图片的宽度和高度，另一个接收一个泛型对象，如下所示：

```
downloadOnly(int width, int height)  
downloadOnly(Y target)
```

这两个方法各自有各自的应用场景，其中downloadOnly(int width, int height)是用于在子线程中下载图片的，而downloadOnly(Y target)是用于在主线程中下载图片的。

那么我们先来看downloadOnly(int width, int height)的用法。当调用了downloadOnly(int width, int height)方法后会立即返回一个FutureTarget对象，然后Glide会在后台开始下载图片文件。接下来我们调用FutureTarget的get()方法就可以去获取下载好的图片文件了，如果此

时图片还没有下载完，那么get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

```
public void downloadImage(View view) {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                String url =  
                    "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
CN7336795473_1920x1080.jpg";  
                final Context context =  
                    getApplicationContext();  
                FutureTarget<File> target =  
                    Glide.with(context)  
                        .load(url)  
                        .downloadOnly(Target.SIZE_ORIGINAL,  
                                     Target.SIZE_ORIGINAL);  
                final File imageFile = target.get();  
                runOnUiThread(new Runnable() {  
                    @Override  
                    public void run() {  
                        Toast.makeText(context,  
                                      imageFile.getPath(),  
                                      Toast.LENGTH_LONG).show();  
                    }  
                });  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }).start();  
}
```

这段代码稍微有一点点长，我带着大家解读一下。首先刚才说了，`downloadOnly(int width, int height)`方法必须要用在子线程当中，因此这里的一步就是new了一个Thread。在子线程当中，我们先获取了一个Application Context，这个时候不能再用Activity作为Context了，因为会有Activity销毁了但子线程还没执行完这种可能出现。

接下来就是Glide的基本用法，只不过将`into()`方法替换成`downloadOnly()`方法。`downloadOnly()`方法会返回一个`FutureTarget`对象，这个时候其实Glide已经开始在后台下载图片了，我们随时都可以调用`FutureTarget`的`get()`方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的`File`对象返回。

最后，我们使用`runOnUiThread()`切回到主线程，然后使用`Toast`将下载好的图片文件路径显示出来。

现在重新运行一下代码，效果如下图所示。



这样我们就能清晰地看出来图片完整的缓存路径是什么了。

之后我们可以使用如下代码去加载这张图片，图片就会立即显示出来，而不用再去网络上请求了：

```
public void loadImage(View view) {  
    String url =  
        "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
        CN7336795473_1920x1080.jpg";  
    Glide.with(this)  
        .load(url)  
        .diskCacheStrategy(DiskCacheStrategy.SOURCE)  
        .into(imageView);  
}
```

需要注意的是，这里必须将硬盘缓存策略指定成 DiskCacheStrategy.SOURCE 或者 DiskCacheStrategy.ALL，否则 Glide 将无法使用我们刚才下载好的图片缓存文件。

现在重新运行一下代码，效果如下图所示。



可以看到，图片的加载和显示是非常快的，因为 Glide 直接使用的是刚才下载好的缓存文件。

那么这个 downloadOnly(int width, int height) 方法的工作原理到底是什么样的呢？我们来简单快速地看一下它的源码吧。

首先在DrawableTypeRequest类当中可以找到定义这个地方，如下所示：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType>  
implements DownloadOptions {  
    ...  
  
    public FutureTarget<File> downloadOnly(int width, int  
height) {  
        return  
getDownloadOnlyRequest().downloadOnly(width, height);  
    }  
  
    private GenericTranscodeRequest<ModelType,  
InputStream, File> getDownloadOnlyRequest() {  
        return optionsApplier.apply(new  
GenericTranscodeRequest<ModelType, InputStream, File>(  
        File.class, this, streamModelLoader,  
InputStream.class, File.class, optionsApplier));  
    }  
}
```

这里会先调用getDownloadOnlyRequest()方法得到一个GenericTranscodeRequest对象，然后再调用它的downloadOnly()方法，代码如下所示：

```
public class GenericTranscodeRequest<ModelType, DataType,  
ResourceType>  
implements DownloadOptions {  
    ...  
  
    public FutureTarget<File> downloadOnly(int width, int  
height) {  
        return getDownloadOnlyRequest().into(width,  
height);  
    }  
}
```

```
    private GenericRequestBuilder<ModelType, DataType, File, File> getDownloadOnlyRequest() {
        ResourceTranscoder<File, File> transcoder =
        UnitTranscoder.get();
        DataLoadProvider<DataType, File> dataLoadProvider =
        glide.buildDataProvider(dataClass, File.class);
        FixedLoadProvider<ModelType, DataType, File, File> fixedLoadProvider =
            new FixedLoadProvider<ModelType, DataType, File, File>(modelLoader, transcoder, dataLoadProvider);
        return optionsApplier.apply(
            new GenericRequestBuilder<ModelType, DataType, File, File>(fixedLoadProvider,
            File.class, this))
            .priority(Priority.LOW)
        .diskCacheStrategy(DiskCacheStrategy.SOURCE)
        .skipMemoryCache(true);
    }
}
```

这里又是调用了一个getDownloadOnlyRequest()方法来构建了一个图片下载的请求，getDownloadOnlyRequest()方法会返回一个GenericRequestBuilder对象，接着调用它的into(width, height)方法，我们继续跟进去瞧一瞧：

```
public FutureTarget<TranscodeType> into(int width, int
height) {
    final RequestFutureTarget<ModelType, TranscodeType>
target =
    new RequestFutureTarget<ModelType,
TranscodeType>(glide.getMainHandler(), width, height);
    glide.getMainHandler().post(new Runnable() {
        @Override
        public void run() {
```

```
        if (!target.isCancelled()) {
            into(target);
        }
    });
    return target;
}
```

那么也就是说，其实这里就是调用了接收Target参数的into()方法，然后Glide就开始执行正常的图片加载逻辑了。那么现在剩下的问题就是，这个RequestFutureTarget中到底处理了些什么逻辑？我们打开它的源码看一看：

```
public class RequestFutureTarget<T, R> implements
FutureTarget<R>, Runnable {

    ...

    @Override
    public R get() throws InterruptedException,
ExecutionException {
        try {
            return doGet(null);
        } catch (TimeoutException e) {
            throw new AssertionError(e);
        }
    }

    @Override
    public R get(long time, TimeUnit timeUnit) throws
InterruptedException, ExecutionException,
TimeoutException {
        return doGet(timeUnit.toMillis(time));
    }

    @Override
    public void getSize(SizeReadyCallback cb) {
        cb.onSizeReady(width, height);
    }
}
```

```
}

@Override
public synchronized void onLoadFailed(Exception e,
Drawable errorDrawable) {
    exceptionReceived = true;
    this.exception = e;
    waiter.notifyAll(this);
}

@Override
public synchronized void onResourceReady(R resource,
GlideAnimation<? super R> glideAnimation) {
    resultReceived = true;
    this.resource = resource;
    waiter.notifyAll(this);
}

private synchronized R doGet(Long timeoutMillis)
throws ExecutionException, InterruptedException,
TimeoutException {
    if (assertBackgroundThread) {
        util.assertBackgroundThread();
    }

    if (isCancelled) {
        throw new CancellationException();
    } else if (exceptionReceived) {
        throw new ExecutionException(exception);
    } else if (resultReceived) {
        return resource;
    }

    if (timeoutMillis == null) {
        waiter.waitForTimeout(this, 0);
    } else if (timeoutMillis > 0) {
        waiter.waitForTimeout(this, timeoutMillis);
    }
}
```

```
        if (Thread.interrupted()) {
            throw new InterruptedException();
        } else if (exceptionReceived) {
            throw new ExecutionException(exception);
        } else if (isCancelled) {
            throw new CancellationException();
        } else if (!resultReceived) {
            throw new TimeoutException();
        }

        return resource;
    }

    static class Waiter {

        public void waitForTimeout(Object toWaitOn, long timeoutMillis) throws InterruptedException {
            toWaitOn.wait(timeoutMillis);
        }

        public void notifyAll(Object toNotify) {
            toNotify.notifyAll();
        }
    }

    ...
}
```

这里我对RequestFutureTarget的源码做了一些精简，我们只看最主要的逻辑就可以了。

刚才我们已经学习过了downloadOnly()方法的基本用法，在调用了downloadOnly()方法之后，再调用FutureTarget的get()方法，就能获取到下载的图片文件了。而downloadOnly()方法返回的FutureTarget对象其实就是这个RequestFutureTarget，因此我们直接来看它的get()方法就行了。

RequestFutureTarget的get()方法中又调用了一个doGet()方法，而doGet()方法才是真正处理具体逻辑的地方。首先在doGet()方法中会判断当前是否是在子线程当中，如果不是的话会直接抛出一个异常。然后下面会判断下载是否已取消、或者已失败，如果是已取消或者已失败的话都会直接抛出一个异常。接下来会根据resultReceived这个变量来判断下载是否已完成，如果这个变量为true的话，就直接把结果进行返回。

那么如果下载还没有完成呢？我们继续往下看，接下来就进入到一个wait()当中，把当前线程给阻塞住，从而阻止代码继续往下执行。这也是为什么downloadOnly(int width, int height)方法要求必须在子线程当中使用，因为它会对当前线程进行阻塞，如果在主线程当中使用的话，那么就会让主线程卡死，从而用户无法进行任何其他操作。

那么现在线程被阻塞住了，什么时候才能恢复呢？答案在onResourceReady()方法中。可以看到，onResourceReady()方法中只有三行代码，第一行把resultReceived赋值成true，说明图片文件已经下载好了，这样下次再调用get()方法时就不会再阻塞线程，而是可以直接将结果返回。第二行把下载好的图片文件赋值到一个全局的resource变量上面，这样doGet()方法就也可以访问到它。第三行notifyAll一下，通知所有wait的线程取消阻塞，这个时候图片文件已经下载好了，因此doGet()方法也就可以返回结果了。

好的，这就是downloadOnly(int width, int height)方法的基本用法和实现原理，那么下面我们来看一下downloadOnly(Y target)方法。

回想一下，其实downloadOnly(int width, int height)方法必须使用在子线程当中，最主要还是因为它在内部帮我们自动创建了一个RequestFutureTarget，是这个RequestFutureTarget要求必须在子线程当中执行。而downloadOnly(Y target)方法则要求我们传入一个自己创建的Target，因此就不受RequestFutureTarget的限制了。

但是downloadOnly(Y target)方法的用法也会相对更复杂一些，因为我们又要自己创建一个Target了，而且这次必须直接去实现最顶层的Target接口，比之前的SimpleTarget和ViewTarget都要复杂不少。

那么下面我们就来实现一个最简单的DownloadImageTarget吧，注意Target接口的泛型必须指定成File对象，这是downloadOnly(Y target)方法要求的，代码如下所示：

```
public class DownloadImageTarget implements Target<File>
{
    private static final String TAG =
"DownloadImageTarget";

    @Override
    public void onStart() {
    }

    @Override
    public void onStop() {
    }

    @Override
    public void onDestroy() {
    }

    @Override
    public void onLoadStarted(Drawable placeholder) {
    }

    @Override
    public void onLoadFailed(Exception e, Drawable errorDrawable) {
    }

    @Override
    public void onResourceReady(File resource,
GlideAnimation<? super File> glideAnimation) {
        Log.d(TAG, resource.getPath());
    }

    @Override
    public void onLoadCleared(Drawable placeholder) {
    }

    @Override
```

```
public void getSize(SizeReadyCallback cb) {
    cb.onSizeReady(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL);
}

@Override
public void setRequest(Request request) {
}

@Override
public Request getRequest() {
    return null;
}

}
```

由于是要直接实现Target接口，因此需要重写的方法非常多。这些方法大多是数Glide加载图片生命周期的一些回调，我们可以不用管它们，其中只有两个方法是必须实现的，一个是getSize()方法，一个是onResourceReady()方法。

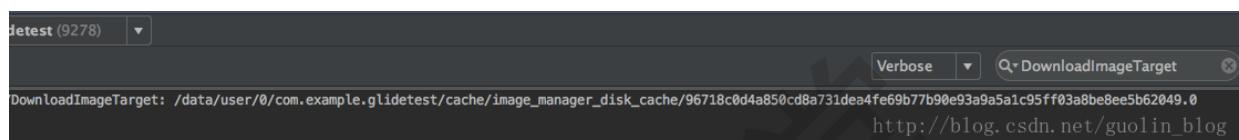
在第二篇Glide源码解析的时候，我带着大家一起分析过，Glide在开始加载图片之前会先计算图片的大小，然后回调到onSizeReady()方法当中，之后才会开始执行图片加载。而这里，计算图片大小的任务就交给了我们了。只不过这是一个最简单的Target实现，我在getSize()方法中就直接回调了Target.SIZE_ORIGINAL，表示图片的原始尺寸。

然后onResourceReady()方法我们就非常熟悉了，图片下载完成之后就会回调到这里，我在这个方法中只是打印了一下下载的图片文件的路径。

这样一个最简单的DownloadImageTarget就定义好了，使用它也非常的简单，我们不用再考虑什么线程的问题了，而是直接把它的实例传入downloadOnly(Y target)方法中即可，如下所示：

```
public void downloadImage(View view) {  
    String url =  
        "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
        CN7336795473_1920x1080.jpg";  
    Glide.with(this)  
        .load(url)  
        .downloadOnly(new DownloadImageTarget());  
}
```

现在重新运行一下代码并点击Download Image按钮，然后观察控制台日志的输出，结果如下图所示。



这样我们就使用了downloadOnly(Y target)方法同样获取到下载的图片文件的缓存路径了。

好的，那么关于downloadOnly()方法我们就学到这里。

listener()方法

今天学习的内容已经够多了，下面我们就以一个简单的知识点结尾吧，Glide回调与监听的最后一部分——listener()方法。

其实listener()方法的作用非常普遍，它可以用来监听Glide加载图片的状态。举个例子，比如说我们刚才使用了preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果Glide加载图片失败了，我该怎样调试错误的原因呢？答案都在listener()方法当中。

首先来看下listener()方法的基本用法吧，不同于刚才几个方法都是要替换into()方法的，listener()是结合into()方法一起使用的，当然也可以结合preload()方法一起使用。最基本的用法如下所示：

```
public void loadImage(View view) {  
    String url =  
        "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
        CN7336795473_1920x1080.jpg";  
}
```

```
Glide.with(this)
    .load(url)
    .listener(new RequestListener<String,
GlideDrawable>() {
        @Override
        public boolean onException(Exception e,
String model, Target<GlideDrawable> target,
        boolean isFirstResource) {
            return false;
        }

        @Override
        public boolean
onResourceReady(GlideDrawable resource, String model,
                Target<GlideDrawable> target, boolean
isFromMemoryCache, boolean isFirstResource) {
            return false;
        }
    })
    .into(imageView);
}
```

这里我们在into()方法之前串接了一个listener()方法，然后实现了一个RequestListener的实例。其中RequestListener需要实现两个方法，一个onResourceReady()方法，一个onException()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调onResourceReady()方法，而当图片加载失败的时候就会回调onException()方法，onException()方法中会将失败的Exception参数传进来，这样我们就可以定位具体失败的原因了。

没错，listener()方法就是这么简单。不过还有一点需要处理，onResourceReady()方法和onException()方法都有一个布尔值的返回值，返回false就表示这个事件没有被处理，还会继续向下传递，返回true就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单的例子，如果我们在RequestListener的onResourceReady()方法中返回了true，那么就不会再回调Target的onResourceReady()方法了。

关于listener()方法的用法就讲这么多，不过还是老规矩，我们再来看一下它的源码是怎么实现的吧。

首先，listener()方法是定义在GenericRequestBuilder类当中的，而我们传入到listener()方法中的实例则会赋值到一个requestListener变量当中，如下所示：

```
public class GenericRequestBuilder<ModelType, DataType,  
    ResourceType, TranscodeType> implements Cloneable {  
  
    private RequestListener<? super ModelType,  
        TranscodeType> requestListener;  
  
    ...  
  
    public GenericRequestBuilder<ModelType, DataType,  
        ResourceType, TranscodeType> listener(  
        RequestListener<? super ModelType,  
        TranscodeType> requestListener) {  
        this.requestListener = requestListener;  
        return this;  
    }  
  
    ...  
}
```

接下来在构建GenericRequest的时候这个变量也会被一起传进去，最后在图片加载完成的时候，我们会看到如下逻辑：

```
public final class GenericRequest<A, T, Z, R> implements  
    Request, SizeReadyCallback,  
    ResourceCallback {  
  
    private RequestListener<? super A, R>  
        requestListener;  
  
    ...  
  
    private void onResourceReady(Resource<?> resource, R  
        result) {
```

```
        boolean isFirstResource = isFirstReadyResource();
        status = Status.COMPLETE;
        this.resource = resource;
        if (requestListener == null ||
!requestListener.onResourceReady(result, model, target,
            loadedFromMemoryCache, isFirstResource))
{
    GlideAnimation<R> animation =
animationFactory.build(loadedFromMemoryCache,
isFirstResource);
    target.onResourceReady(result, animation);
}
notifyLoadSuccess();
}
...
}
```

可以看到，这里在第11行会先回调requestListener的onResourceReady()方法，只有当这个onResourceReady()方法返回false的时候，才会继续调用Target的onResourceReady()方法，这也就是listener()方法的实现原理。

另外一个onException()方法的实现机制也是一模一样的，代码同样是在GenericRequest类中，如下所示：

```
public final class GenericRequest<A, T, Z, R> implements
Request, SizeReadyCallback,
ResourceCallback {
    ...
    @Override
    public void onException(Exception e) {
        status = Status.FAILED;
        if (requestListener == null ||
            !requestListener.onException(e, model,
target, isFirstReadyResource())) {
            setErrorPlaceholder(e);
        }
    }
}
```

```
}
```

```
...
```

```
}
```

可以看到，这里会在第9行回调requestListener的onException()方法，只有在onException()方法返回false的情况下才会继续调用setErrorPlaceholder()方法。也就是说，如果我们在onException()方法中返回了true，那么Glide请求中使用error(int resourceId)方法设置的异常占位图就失效了。

这样我们也就将listener()方法的全部实现原理都分析完了。

好了，关于Glide回调与监听方面的内容今天就讲到这里，这一篇文章的内容非常充实，希望大家都能好好掌握。

3.6 Glide源码学习六：图片变换

Glide的这个框架的功能实在是太强大了，它所能做的事情远远不止于目前我们所学的这些。因此，今天我们就再来学习一个新的功能模块，并且是一个非常重要的模块——Glide的图片变化功能。

一个问题

在正式开始学习Glide的图片变化功能之前，我们先来看一个问题，这个问题可能有不少人都在使用Glide的时候都遇到过，正好在本篇内容的主题之下我们顺带着将这个问题给解决了。

首先我们尝试使用Glide来加载一张图片，图片URL地址是：

```
https://www.baidu.com/img/bd_logo1.png
```

这是百度首页logo的一张图片，图片尺寸是540*258像素。

接下来我们编写一个非常简单的布局文件，如下所示：

```
<LinearLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Image"
        android:onClick="loadImage"
        />
    <ImageView
        android:id="@+id/image_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

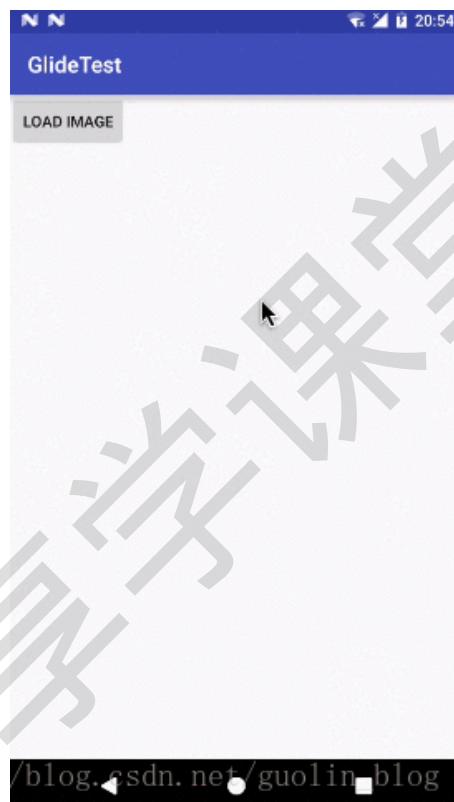
布局文件中只有一个按钮和一个用于显示图片的ImageView。注意，
ImageView的宽和高这里设置的都是wrap_content。

然后编写如下的代码来加载图片：

```
public class MainActivity extends AppCompatActivity {
    ImageView imageView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        imageView = (ImageView)
        findViewById(R.id.image_view);
    }
}
```

```
public void loadImage(View view) {  
    String url =  
    "https://www.baidu.com/img/bd_logo1.png";  
    Glide.with(this)  
        .load(url)  
        .into(imageView);  
}  
}
```

这些简单的代码对于现在的你而言应该都是小儿科了，相信我也不用再做什么解释。现在运行一下程序并点击加载图片按钮，效果如下图所示。



图片是正常加载出来了，不过大家有没有发现一个问题。百度这张logo图片的尺寸只有540*258像素，但是我的手机的分辨率却是1080*1920像素，而我们将ImageView的宽高设置的都是wrap_content，那么图片的宽度应该只有手机屏幕宽度的一半而已，但是这里却充满了全屏，这是为什么呢？

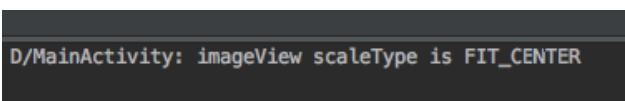
如果你之前也被这个问题困扰过，那么恭喜，本篇文章正是你所需要的。之所以会出现这个现象，就是因为Glide的图片变换功能所导致的。那么接下来我们会先分析如何解决这个问题，然后再深入学习Glide图片变化的更多功能。

稍微对Android有点了解的人应该都知道ImageView有scaleType这个属性，但是可能大多数人却不知道，如果在没有指定scaleType属性的情况下，ImageView默认的scaleType是什么？

这个问题如果直接问我，我也答不上来。不过动手才是检验真理的唯一标准，想知道答案，自己动手试一下就知道了。

```
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "MainActivity";  
  
    ImageView imageView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        imageView = (ImageView)  
findviewById(R.id.image_view);  
        Log.d(TAG, "imageView scaleType is " +  
imageView.getScaleType());  
    }  
  
    ...  
}
```

可以看到，我们在onCreate()方法中打印了ImageView默认的scaleType，然后重新运行一下程序，结果如下图所示：



由此我们可以得知，在没有明确指定的情况下，ImageView默认的scaleType是FIT_CENTER。

有了这个前提条件，我们就可以继续去分析Glide的源码了。

回顾一下我们分析过的into()方法，它是在GenericRequestBuilder类当中的，代码如下所示：

```
public Target<TranscodeType> into(ImageView view) {
    Util.assertMainThread();
    if (view == null) {
        throw new IllegalArgumentException("You must pass
in a non null view");
    }
    if (!isTransformationSet && view.getScaleType() !=
null) {
        switch (view.getScaleType()) {
            case CENTER_CROP:
                applyCenterCrop();
                break;
            case FIT_CENTER:
            case FIT_START:
            case FIT_END:
                applyFitCenter();
                break;
            // $CASES-OMITTED$
            default:
                // Do nothing.
        }
    }
    return into(glide.buildImageViewTarget(view,
transcodeClass));
}
```

还记得我们当初分析这段代码的时候，直接跳过前面的所有代码，直奔最后一行。因为那个时候我们的主要任务是分析Glide的主线执行流程，而不去仔细阅读它的细节，但是现在我们是时候应该阅读一下细节了。

可以看到，这里在第7行会进行一个switch判断，如果ImageView的scaleType是CENTER_CROP，则会去调用applyCenterCrop()方法，如果scaleType是FIT_CENTER、FIT_START或FIT_END，则会去调用applyFitCenter()方法。这里的applyCenterCrop()和applyFitCenter()方法其实就是在向Glide的加载流程中添加了一个图片变换操作，具体的源码我们就不跟进去看了。

那么现在我们就基本清楚了，由于ImageView默认的scaleType是FIT_CENTER，因此会自动添加一个FitCenter的图片变换，而在这个图片变换过程中做了某些操作，导致图片充满了全屏。

那么我们该如何解决这个问题呢？最直白的一种办法就是看着源码来改。当ImageView的scaleType是CENTER_CROP、FIT_CENTER、FIT_START或FIT_END时不是会自动添加一个图片变换操作吗？那我们把scaleType改成其他值不就可以了。ImageView的scaleType可选值还有CENTER、CENTER_INSIDE、FIT_XY等。这当然是一种解决方案，不过只能说是一种比较笨的解决方案，因为我们为了解决这个问题而去改动了ImageView原有的scaleType，那如果你真的需要ImageView的scaleType为CENTER_CROP或FIT_CENTER时可能就傻眼了。

上面只是我们通过分析源码得到的一种解决方案，并不推荐大家使用。实际上，Glide给我们提供了专门的API来添加和取消图片变换，想要解决这个问题只需要使用如下代码即可：

```
Glide.with(this)
    .load(url)
    .dontTransform()
    .into(imageview);
```

可以看到，这里调用了一个dontTransform()方法，表示让Glide在加载图片的过程中不进行图片变换，这样刚才调用的applyCenterCrop()、applyFitCenter()就统统无效了。

现在我们重新运行一下代码，效果如下图所示：



这样图片就只会占据半个屏幕的宽度了，说明我们的代码奏效了。

但是使用`dontTransform()`方法存在着一个问题，就是调用这个方法之后，所有的图片变换操作就全部失效了，那如果我有一些图片变换操作是必须要执行的该怎么办呢？不用担心，总归是有办法的，这种情况下我们只需要借助`override()`方法强制将图片尺寸指定成原始大小就可以了，代码如下所示：

```
Glide.with(this)
    .load(url)
    .override(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL)
    .into(imageView);
```

通过`override()`方法将图片的宽和高都指定成`Target.SIZE_ORIGINAL`，问题同样被解决了。程序的最终运行结果和上图是完全一样的，我就不再重新截图了。

由此我们可以看出，之所以会出现这个问题，和Glide的图片变换功能是撇不开关系的。那么也是通过这个问题，我们对Glide的图片变换有了一个最基本的认识。接下来，就让我们正式开始进入本篇文章的正题吧。

图片变换的基本用法

顾名思义，图片变换的意思就是说，Glide从加载了原始图片到最终展示给用户之前，又进行了一些变换处理，从而能够实现一些更加丰富的图片效果，如图片圆角化、圆形化、模糊化等等。

添加图片变换的用法非常简单，我们只需要调用transform()方法，并将想要执行的图片变换操作作为参数传入transform()方法即可，如下所示：

```
Glide.with(this)
    .load(url)
    .transform(...)
    .into(imageview);
```

至于具体要进行什么样的图片变换操作，这个通常都是需要我们自己来写的。不过Glide已经内置了两种图片变换操作，我们可以直接拿来使用，一个是CenterCrop，一个是FitCenter。

但这两种内置的图片变换操作其实都不需要使用transform()方法，Glide为了方便我们使用直接提供了现成的API：

```
Glide.with(this)
    .load(url)
    .centerCrop()
    .into(imageview);

Glide.with(this)
    .load(url)
    .fitCenter()
    .into(imageview);
```

当然，centerCrop()和fitCenter()方法其实也只是对transform()方法进行了一层封装而已，它们背后的源码仍然还是借助transform()方法来实现的，如下所示：

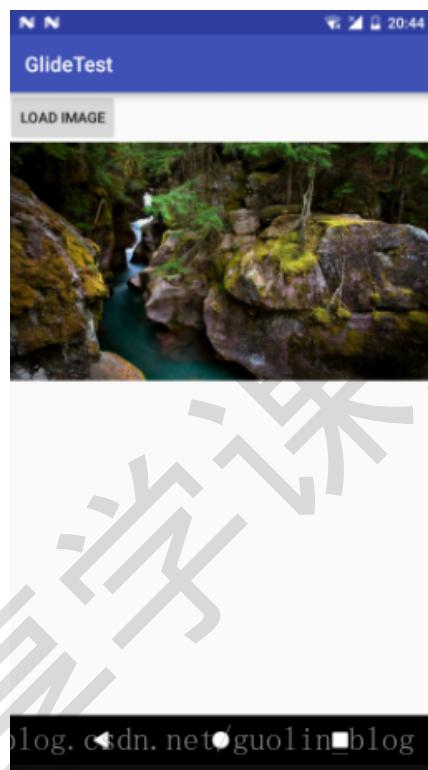
```
public class DrawableRequestBuilder<ModelType>
    extends GenericRequestBuilder<ModelType,
    ImageVideoWrapper, GifBitmapWrapper, GlideDrawable>
```

```
    implements BitmapOptions, DrawableOptions {  
    ...  
  
    /**  
     * Transform {@link GlideDrawable}s using {@link  
     * com.bumptech.glide.load.resource.bitmap.CenterCrop}.  
     *  
     * @see #fitCenter()  
     * @see #transform(BitmapTransformation...)  
     * @see #bitmapTransform(Transformation[])  
     * @see #transform(Transformation[])  
     *  
     * @return This request builder.  
     */  
    @SuppressWarnings("unchecked")  
    public DrawableRequestBuilder<ModelType> centerCrop()  
{  
    return transform(glide.getDrawableCenterCrop());  
}  
  
    /**  
     * Transform {@link GlideDrawable}s using {@link  
     * com.bumptech.glide.load.resource.bitmap.FitCenter}.  
     *  
     * @see #centerCrop()  
     * @see #transform(BitmapTransformation...)  
     * @see #bitmapTransform(Transformation[])  
     * @see #transform(Transformation[])  
     *  
     * @return This request builder.  
     */  
    @SuppressWarnings("unchecked")  
    public DrawableRequestBuilder<ModelType> fitCenter()  
{  
    return transform(glide.getDrawableFitCenter());  
}  
...  
}
```

}

那么这两种内置的图片变换操作到底能实现什么样的效果呢？FitCenter的效果其实刚才我们已经见识过了，就是会将图片按照原始的长宽比充满全屏。那么CenterCrop又是什么样的效果呢？我们来动手试一下就知道了。

为了让效果更加明显，这里我就不使用百度首页的Logo图了，而是换成必应首页的一张美图。在不应用任何图片变换的情况下，使用Glide加载必应这张图片效果如下所示。



现在我们添加一个CenterCrop的图片变换操作，代码如下：

```
String url =  
"http://cn.bing.com/az/hprichbg/rb/AvalancheCreek_ROW1117  
3354624_1920x1080.jpg";  
Glide.with(this)  
    .load(url)  
    .centerCrop()  
    .into(imageView);
```

重新运行一下程序并点击加载图片按钮，效果如下图所示。

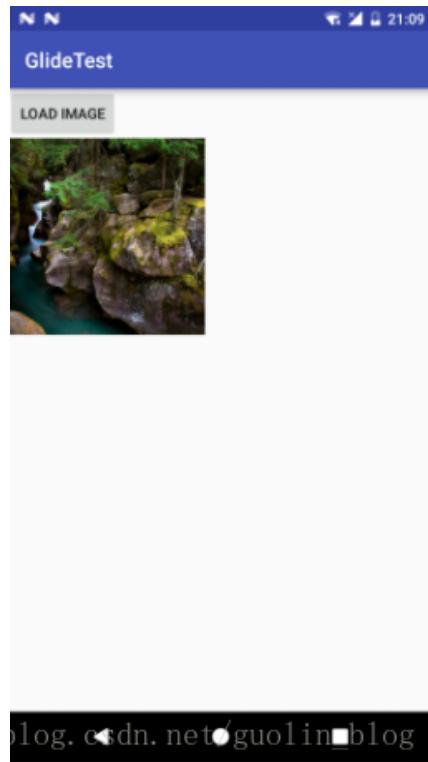


可以看到，现在展示的图片是对原图的中心区域进行裁剪后得到的图片。

另外，centerCrop()方法还可以配合override()方法来实现更加丰富的效果，比如指定图片裁剪的比例：

```
String url =  
    "http://cn.bing.com/az/hprichbg/rb/AvalancheCreek_ROW1117  
3354624_1920x1080.jpg";  
Glide.with(this)  
    .load(url)  
    .override(500, 500)  
    .centerCrop()  
    .into(imageview);
```

可以看到，这里我们将图片的尺寸设定为500*500像素，那么裁剪的比例也就变成1：1了，现在重新运行一下程序，效果如下图所示。



这样我们就把Glide内置的图片变换接口的用法都掌握了。不过不得不说，Glide内置的图片变换接口功能十分单一且有限，完全没有办法满足我们平时的开发需求。因此，掌握自定义图片变换功能就显得尤为重要了。

不过，在正式开始学习自定义图片变换功能之前，我们先来探究一下CenterCrop这种图片变换的源码，理解了它的源码我们再来进行自定义图片变换就能更加得心应手了。

源码分析

那么就话不多说，我们直接打开CenterCrop类来看一下它的源码吧，如下所示：

```
public class CenterCrop extends BitmapTransformation {  
  
    public CenterCrop(Context context) {  
        super(context);  
    }  
  
    public CenterCrop(BitmapPool bitmapPool) {  
        super(bitmapPool);  
    }  
}
```

```
// Bitmap doesn't implement equals, so == and .equals  
// are equivalent here.  
  
@SuppressWarnings("PMD.CompareObjectsWithEquals")  
@Override  
protected Bitmap transform(BitmapPool pool, Bitmap  
toTransform, int outwidth, int outHeight) {  
    final Bitmap toReuse = pool.get(outwidth,  
    outHeight, toTransform.getConfig() != null  
        ? toTransform.getConfig() :  
    Bitmap.Config.ARGB_8888);  
    Bitmap transformed =  
TransformationUtils.centerCrop(toReuse, toTransform,  
outwidth, outHeight);  
    if (toReuse != null && toReuse != transformed &&  
!pool.put(toReuse)) {  
        toReuse.recycle();  
    }  
    return transformed;  
}  
  
@Override  
public String getId() {  
    return  
"CenterCrop.com.bumptech.glide.load.resource.bitmap";  
}  
}
```

这段代码并不长，但是我还是要划下重点，这样大家看起来的时候会更加轻松。

首先，CenterCrop是继承自BitmapTransformation的，这个是重中之重，因为整个图片变换功能都是建立在这个继承结构基础上的。

接下来CenterCrop中最重要的就是transform()方法，其他的方法我们可以暂时忽略。transform()方法中有四个参数，每一个都很重要，我们一一解读下。第一个参数pool，这是Glide中的一个Bitmap缓存池，用于对Bitmap对象进行重用，否则每次图片变换都重新创建Bitmap对象将会非常消耗内存。第二个参数toTransform，这是原始图片的Bitmap对

象，我们就是要对它来进行图片变换。第三和第四个参数比较简单，分别代表图片变换后的宽度和高度，其实也就是override()方法中传入的宽和高的值了。

下面我们来看一下transform()方法的细节，首先第一行就从Bitmap缓存池中尝试获取一个可重用的Bitmap对象，然后把这个对象连同toTransform、outWidth、outHeight参数一起传入到了TransformationUtils.centerCrop()方法当中。那么我们就跟进去来看一下这个方法的源码，如下所示：

```
public final class TransformationUtils {  
    ...  
  
    public static Bitmap centerCrop(Bitmap recycled,  
        Bitmap toCrop, int width, int height) {  
        if (toCrop == null) {  
            return null;  
        } else if (toCrop.getWidth() == width &&  
        toCrop.getHeight() == height) {  
            return toCrop;  
        }  
        // From ImageView/Bitmap.createScaledBitmap.  
        final float scale;  
        float dx = 0, dy = 0;  
        Matrix m = new Matrix();  
        if (toCrop.getWidth() * height > width *  
        toCrop.getHeight()) {  
            scale = (float) height / (float)  
        toCrop.getHeight();  
            dx = (width - toCrop.getWidth() * scale) *  
        0.5f;  
        } else {  
            scale = (float) width / (float)  
        toCrop.getWidth();  
            dy = (height - toCrop.getHeight() * scale) *  
        0.5f;  
        }  
        m.setScale(scale, scale);
```

```
m.postTranslate((int) (dx + 0.5f), (int) (dy +  
0.5f));  
  
    final Bitmap result;  
    if (recycled != null) {  
        result = recycled;  
    } else {  
        result = Bitmap.createBitmap(width, height,  
getSafeConfig(toCrop));  
    }  
  
    // we don't add or remove alpha, so keep the  
alpha setting of the Bitmap we were given.  
    TransformationUtils.setAlpha(toCrop, result);  
  
    Canvas canvas = new Canvas(result);  
    Paint paint = new Paint(PAINT_FLAGS);  
    canvas.drawBitmap(toCrop, m, paint);  
    return result;  
}  
  
...  
}
```

这段代码就是整个图片变换功能的核心代码了。可以看到，第5-9行主要是先做了一些校验，如果原图为空，或者原图的尺寸和目标裁剪尺寸相同，那么就放弃裁剪。接下来第11-22行是通过数学计算来算出画布的缩放的比例以及偏移值。第24-29行是判断缓存池中取出的Bitmap对象是否为空，如果不为空就可以直接使用，为空则要创建一个新的Bitmap对象。第32行是将原图Bitmap对象的alpha值复制到裁剪Bitmap对象上面。最后第34-37行是裁剪Bitmap对象进行绘制，并将最终的结果进行返回。全部的逻辑就是这样，总体来说还是比较简单的，可能也就是数学计算那边需要稍微动下脑筋。

那么现在得到了裁剪后的Bitmap对象，我们再回到CenterCrop当中，你会看到，在最终返回这个Bitmap对象之前，还会尝试将复用的Bitmap对象重新放回到缓存池当中，以便下次继续使用。

好的，这样我们就将CenterCrop图片变换的工作原理完整地分析了一遍，FitCenter的源码也是基本类似的，这里就不再重复分析了。了解了这些内容之后，接下来我们就可以开始学习自定义图片变换功能了。

自定义图片变换

Glide给我们定制好了一个图片变换的框架，大致的流程是我们可以获取到原始的图片，然后对图片进行变换，再将变换完成后的图片返回给Glide，最终由Glide将图片显示出来。理论上，在对图片进行变换这个步骤中我们可以进行任何的操作，你想对图片怎么样都可以。包括圆角化、圆形化、黑白化、模糊化等等，甚至你将原图片完全替换成另外一张图都是可以的。

但是这里显然我不可能向大家演示所有图片变换的可能，图片变换的可能性也是无限的。因此这里我们就选择一种常用的图片变换效果来进行自定义吧——对图片进行圆形化变换。

图片圆形化的功能现在在手机应用中非常常见，比如手机QQ就会将用户的头像进行圆形化变换，从而使得界面变得更加好看。

自定义图片变换功能的实现逻辑比较固定，我们刚才看过CenterCrop的源码之后，相信你已经基本了解整个自定义的过程了。其实就是自定义一个类让它继承自BitmapTransformation，然后重写transform()方法，并在这里去实现具体的图片变换逻辑就可以了。一个空的图片变换实现大概如下所示：

```
public class CircleCrop extends BitmapTransformation {

    public CircleCrop(Context context) {
        super(context);
    }

    public CircleCrop(BitmapPool bitmapPool) {
        super(bitmapPool);
    }

    @Override
    public String getId() {
```

```
        return "com.example.glideTest.CircleCrop";  
    }  
  
    @Override  
    protected Bitmap transform(BitmapPool pool, Bitmap  
toTransform, int outwidth, int outHeight) {  
        return null;  
    }  
}
```

这里有一点需要注意，就是getId()方法中要求返回一个唯一的字符串来作为id，以和其他的图片变换做区分。通常情况下，我们直接返回当前类的完整类名就可以了。

另外，这里我们选择继承BitmapTransformation还有一个限制，就是只能对静态图进行图片变换。当然，这已经足够覆盖日常95%以上的开发需求了。如果你有特殊的需求要对GIF图进行图片变换，那就得去自己实现Transformation接口才就可以了。不过这个就非常复杂了，不在我们今天的讨论范围。

好了，那么我们继续实现对图片进行圆形化变换的功能，接下来只需要在transform()方法中去做具体的逻辑实现就可以了，代码如下所示：

```
public class CircleCrop extends BitmapTransformation {  
  
    public CircleCrop(Context context) {  
        super(context);  
    }  
  
    public CircleCrop(BitmapPool bitmapPool) {  
        super(bitmapPool);  
    }  
  
    @Override  
    public String getId() {  
        return "com.example.glideTest.CircleCrop";  
    }
```

```
    @Override
    protected Bitmap transform(BitmapPool pool, Bitmap
toTransform, int outwidth, int outHeight) {
        int diameter = Math.min(toTransform.getWidth(),
toTransform.getHeight());

        final Bitmap toReuse = pool.get(outwidth,
outHeight, Bitmap.Config.ARGB_8888);
        final Bitmap result;
        if (toReuse != null) {
            result = toReuse;
        } else {
            result = Bitmap.createBitmap(diameter,
diameter, Bitmap.Config.ARGB_8888);
        }

        int dx = (toTransform.getWidth() - diameter) / 2;
        int dy = (toTransform.getHeight() - diameter) /
2;
        Canvas canvas = new Canvas(result);
        Paint paint = new Paint();
        BitmapShader shader = new
BitmapShader(toTransform, BitmapShader.TileMode.CLAMP,
BitmapShader.TileMode.CLAMP);
        if (dx != 0 || dy != 0) {
            Matrix matrix = new Matrix();
            matrix.setTranslate(-dx, -dy);
            shader.setLocalMatrix(matrix);
        }
        paint.setShader(shader);
        paint.setAntiAlias(true);
        float radius = diameter / 2f;
        canvas.drawCircle(radius, radius, radius, paint);

        if (toReuse != null && !pool.put(toReuse)) {
            toReuse.recycle();
        }
    }
}
```

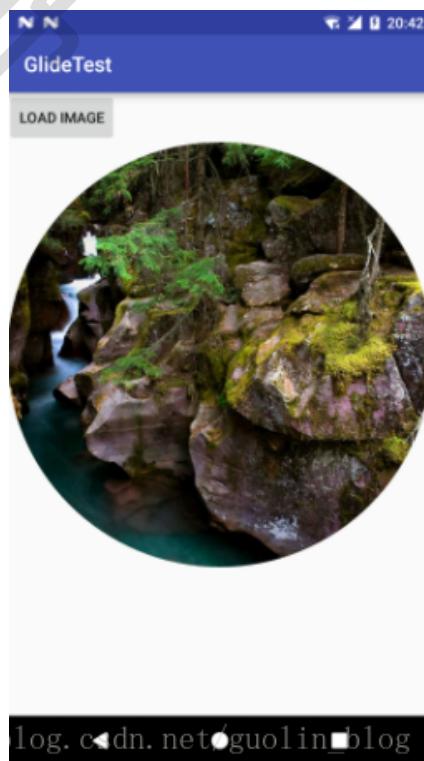
```
    return result;
}
}
```

下面我对transform()方法中的逻辑做下简单的解释。首先第18行先算出原图宽度和高度中较小的值，因为对图片进行圆形化变换肯定要以较小的那个值作为直径来进行裁剪。第20-26行则和刚才一样，从Bitmap缓存池中尝试获取一个Bitmap对象来进行重用，如果没有可重用的Bitmap对象的话就创建一个。第28-41行是具体进行圆形化变换的部分，这里算出了画布的偏移值，并且根据刚才得到的直径算出半径来进行画圆。最后，尝试将复用的Bitmap对象重新放回到缓存池当中，并将圆形化变换后的Bitmap对象进行返回。

这样，一个自定义图片变换的功能就写好了，那么现在我们就来尝试使用一下它吧。使用方法非常简单，刚才已经介绍过了，就是把这个自定义图片变换的实例传入到transform()方法中即可，如下所示：

```
Glide.with(this)
    .load(url)
    .transform(new CircleCrop(this))
    .into(imageview);
```

现在我们重新运行一下程序，效果如下图所示。



更多图片变换功能

虽说Glide的图片变换功能框架已经很强大了，使得我们可以轻松地自定义图片变换效果，但是如果每一种图片变换都要我们自己去写还是蛮吃力的。事实上，确实也没有必要完全靠自己去实现各种各样的图片变换效果，因为大多数的图片变换都是比较通用的，各个项目会用到的效果都差不多，我们每一个都自己去重新实现无异于重复造轮子。

也正是因此，网上出现了很多Glide的图片变换开源库，其中做的最出色的应该要数glide-transformations这个库了。它实现了很多通用的图片变换效果，如裁剪变换、颜色变换、模糊变换等等，使得我们可以非常轻松地进行各种各样的图片变换。

glide-transformations的项目主页地址是

<https://github.com/wasabeef/glide-transformations>

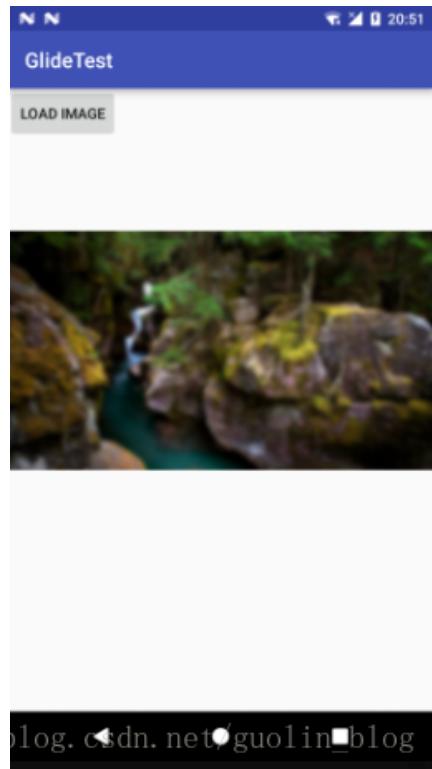
下面我们就来体验一下这个库的强大功能吧。首先需要将这个库引入到我们的项目当中，在app/build.gradle文件当中添加如下依赖：

```
dependencies {  
    compile 'jp.wasabeef:glide-transformations:2.0.2'  
}
```

现在如果我想对图片进行模糊化处理，那么就可以使用glide-transformations库中的BlurTransformation这个类，代码如下所示：

```
Glide.with(this)  
    .load(url)  
    .bitmapTransform(new BlurTransformation(this))  
    .into(imageview);
```

注意这里我们调用的是bitmapTransform()方法而不是transform()方法，因为glide-transformations库都是专门针对静态图片变换来进行设计的。现在重新运行一下程度，效果如下图所示。

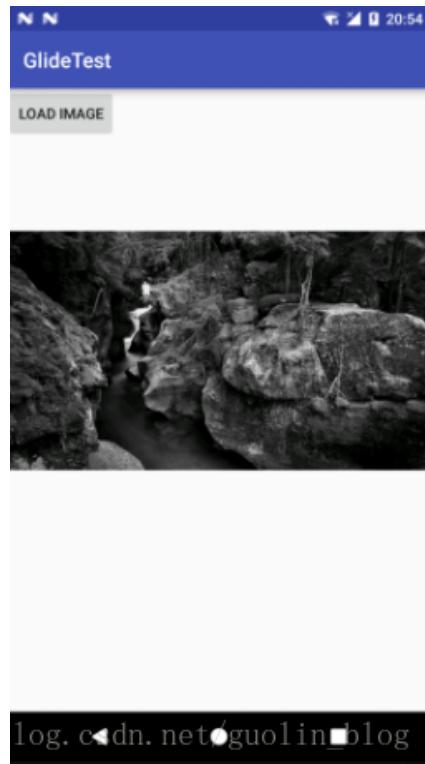


没错，我们就这样轻松地实现模糊化的效果了。

接下来我们再试一下图片黑白化的效果，使用的是
GrayscaleTransformation这个类，代码如下所示：

```
Glide.with(this)
    .load(url)
    .bitmapTransform(new GrayscaleTransformation(this))
    .into(imageview);
```

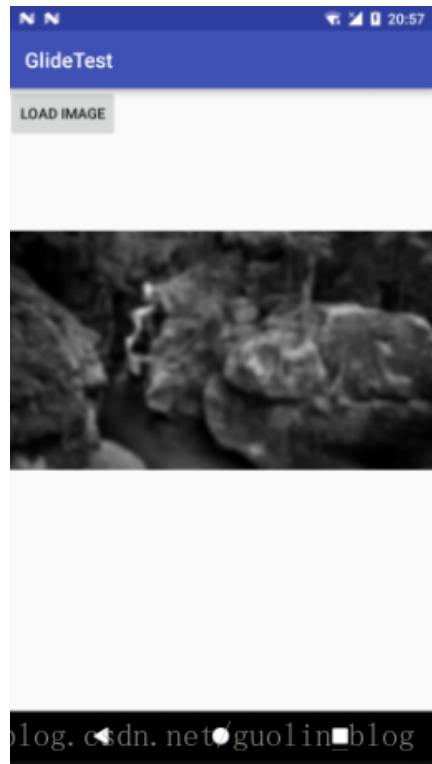
现在重新运行一下程度，效果如下图所示。



而且我们还可以将多个图片变换效果组合在一起使用，比如同时执行模糊化和黑白化的变换：

```
Glide.with(this)
    .load(url)
    .bitmapTransform(new BlurTransformation(this), new
GrayscaleTransformation(this))
    .into(imageview);
```

可以看到，同时执行多种图片变换的时候，只需要将它们都传入到 bitmapTransform()方法中即可。现在重新运行一下程序，效果如下图所示。



当然，这些只是glide-transformations库的一小部分功能而已，更多的图片变换效果你可以到它的GitHub项目主页去学习，所有变换的用法都是这么简单哦。

3.7 Glide源码学习七：自定义模块功能

自定义模块的基本用法

学到这里相信你已经知道，Glide的用法是非常非常简单的，大多数情况下，我们想要实现的图片加载效果只需要一行代码就能解决了。但是Glide过于简洁的API也造成了一个问题，就是如果我们想要更改Glide的某些默认配置项应该怎么操作呢？很难想象如何将更改Glide配置项的操作串联到一行经典的Glide图片加载语句中当中吧？没错，这个时候就需要用到自定义模块功能了。

自定义模块功能可以将更改Glide配置，替换Glide组件等操作独立出来，使得我们能轻松地对Glide的各种配置进行自定义，并且又和Glide的图片加载逻辑没有任何交集，这也是一种低耦合编程方式的体现。那么接下来我们就学习一下自定义模块的基本用法。

首先需要定义一个我们自己的模块类，并让它实现GlideModule接口，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
    }
}
```

可以看到，在MyGlideModule类当中，我们重写了applyOptions()和registerComponents()方法，这两个方法分别就是用来更改Glide和配置以及替换Glide组件的。我们待会儿只需要在这两个方法中加入具体的逻辑，就能实现更改Glide配置或者替换Glide组件的功能了。

不过，目前Glide还无法识别我们自定义的MyGlideModule，如果想要让它生效，还得在AndroidManifest.xml文件当中加入如下配置才行：

```
<manifest>
    ...
<application>
    <meta-data
        android:name="com.example.glidetest.MyGlideModule"
        android:value="GlideModule" />
    ...
</application>
</manifest>
```

在标签中加入一个meta-data配置项，其中android:name指定成我们自定义的MyGlideModule的完整路径， android:value必须指定成 GlideModule，这个是固定值。

这样的话，我们就将Glide自定义模块的功能完成了，是不是非常简单？现在Glide已经能够识别我们自定义的这个MyGlideModule了，但是在编写具体的功能之前，我们还是按照老规矩阅读一下源码，从源码的层面上来分析一下， Glide到底是如何识别出这个自定义的MyGlideModule的。

自定义模块的原理

显然我们已经用惯了Glide.with(context).load(url).into(imageView)这样一行简洁的Glide图片加载语句，但是我们好像从来没有注意过Glide这个类本身的实例。然而事实上， Glide类确实是有创建实例的，只不过是在内部由Glide自动帮我们创建和管理了，对于开发者而言，大多数情况下是不用关心它的，只需要调用它的静态方法就可以了。

那么Glide的实例到底是在哪里创建的呢？我们来看下Glide类中的get()方法的源码，如下所示：

```
public class Glide {  
  
    private static volatile Glide glide;  
  
    ...  
  
    public static Glide get(Context context) {  
        if (glide == null) {  
            synchronized (Glide.class) {  
                if (glide == null) {  
                    Context applicationContext =  
context.getApplicationContext();  
                    List<GlideModule> modules = new  
ManifestParser(applicationContext).parse();  
                    GlideBuilder builder = new  
GlideBuilder(applicationContext);  
                    for (GlideModule module : modules) {  
                        if (module.isManifestParsingEnabled()) {  
                            module.  
...  
                    }  
                }  
            }  
        }  
        return glide;  
    }  
}
```

```
        module.applyOptions(applicationContext, builder);
    }
    glide = builder.createGlide();
    for (GlideModule module : modules) {
        module.registerComponents(applicationContext, glide);
    }
}
return glide;
}

...
}
```

我们来仔细看一下上面这段代码。首先这里使用了一个单例模式来获取 Glide对象的实例，可以看到，这是一个非常典型的双重锁模式。然后在第12行，调用ManifestParser的parse()方法去解析AndroidManifest.xml文件中的配置，实际上就是将AndroidManifest中所有值为GlideModule的meta-data配置读取出来，并将相应的自定义模块实例化。由于你可以自定义任意多个模块，因此这里我们将会得到一个GlideModule的List集合。

接下来在第13行创建了一个GlideBuilder对象，并通过一个循环调用了每一个GlideModule的applyOptions()方法，同时也把GlideBuilder对象作为参数传入到这个方法中。而applyOptions()方法就是我们可以加入自己的逻辑的地方了，虽然目前为止我们还没有编写任何逻辑。

再往下的一步就非常关键了，这里调用了GlideBuilder的createGlide()方法，并返回了一个Glide对象。也就是说，Glide对象的实例就是在这里创建的了，那么我们跟到这个方法当中瞧一瞧：

```
public class GlideBuilder {
    private final Context context;

    private Engine engine;
```

```
private BitmapPool bitmapPool;
private MemoryCache memoryCache;
private ExecutorService sourceService;
private ExecutorService diskCacheService;
private DecodeFormat decodeFormat;
private DiskCache.Factory diskCacheFactory;

...
Glide createGlide() {
    if (sourceService == null) {
        final int cores = Math.max(1,
Runtime.getRuntime().availableProcessors());
        sourceService = new
FifoPriorityThreadPoolExecutor(cores);
    }
    if (diskCacheService == null) {
        diskCacheService = new
FifoPriorityThreadPoolExecutor(1);
    }
    MemorySizeCalculator calculator = new
MemorySizeCalculator(context);
    if (bitmapPool == null) {
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.HONEYCOMB) {
            int size =
calculator.getBitmapPoolSize();
            bitmapPool = new LruBitmapPool(size);
        } else {
            bitmapPool = new BitmapPoolAdapter();
        }
    }
    if (memoryCache == null) {
        memoryCache = new
LruResourceCache(calculator.getMemoryCacheSize());
    }
    if (diskCacheFactory == null) {
```

```
        diskCacheFactory = new
InternalCacheDiskCacheFactory(context);
    }
    if (engine == null) {
        engine = new Engine(memoryCache,
diskCacheFactory, diskCacheService, sourceService);
    }
    if (decodeFormat == null) {
        decodeFormat = DecodeFormat.DEFAULT;
    }
    return new Glide(engine, memoryCache, bitmapPool,
context, decodeFormat);
}
}
```

这个方法中会创建BitmapPool、MemoryCache、DiskCache、DecodeFormat等对象的实例，并在最后一行创建一个Glide对象的实例，然后将前面创建的这些实例传入到Glide对象当中，以供后续的图片加载操作使用。

但是大家有没有注意到一个细节，createGlide()方法中创建任何对象的时候都做了一个空检查，只有在对象为空的时候才会去创建它的实例。也就是说，如果我们在applyOptions()方法中提前就给这些对象初始化并赋值，那么在createGlide()方法中就不会再去重新创建它们的实例了，从而也就实现了更改Glide配置的功能。关于这个功能我们待会儿会进行具体的演示。

现在继续回到Glide的get()方法中，得到了Glide对象的实例之后，接下来又通过一个循环调用了每一个GlideModule的registerComponents()方法，在这里我们可以加入替换Glide的组件的逻辑。

好了，这就是Glide自定义模块的全部工作原理。了解了它的工作原理之后，接下来所有的问题就集中在我到底如何在applyOptions()和registerComponents()这两个方法中加入具体的逻辑了，下面我们马上就开始学习一下。

更改Glide配置

刚才在分析自定义模式工作原理的时候其实就已经提到了，如果想要更改Glide的默认配置，其实只需要在applyOptions()方法中提前将Glide的配置项进行初始化就可以了。那么Glide一共有哪些配置项呢？这里我给大家做了一个列举：

- setMemoryCache() 用于配置Glide的内存缓存策略，默认配置是LruResourceCache。
- setBitmapPool() 用于配置Glide的Bitmap缓存池，默认配置是LruBitmapPool。
- setDiskCache() 用于配置Glide的硬盘缓存策略，默认配置是InternalCacheDiskCacheFactory。
- setDiskCacheService() 用于配置Glide读取缓存中图片的异步执行器，默认配置是FifoPriorityThreadPoolExecutor，也就是先入先出原则。
- setResizeService() 用于配置Glide读取非缓存中图片的异步执行器，默认配置也是FifoPriorityThreadPoolExecutor。
- setDecodeFormat() 用于配置Glide加载图片的解码模式，默认配置是RGB_565。

其实Glide的这些默认配置都非常科学且合理，使用的缓存算法也都是效率极高的，因此在绝大多数情况下我们并不需要去修改这些默认配置，这也是Glide用法能如此简洁的一个原因。

但是Glide科学的默认配置并不影响我们去学习自定义Glide模块的功能，因此总有某些情况下，默认的配置可能将无法满足你，这个时候就需要我们自己动手来修改默认配置了。

下面就通过具体的实例来看一下吧。刚才说到，Glide默认的硬盘缓存策略使用的是InternalCacheDiskCacheFactory，这种缓存会将所有Glide加载的图片都存储到当前应用的私有目录下。这是一种非常安全的做法，但同时这种做法也造成了一些不便，因为私有目录下即使是开发者自己也是无法查看的，如果我想要去验证一下图片到底有没有成功缓存下来，这就有点不太好办了。

这种情况下，就非常适合使用自定义模块来更改Glide的默认配置。我们完全可以自己去实现DiskCache.Factory接口来自定义一个硬盘缓存策略，不过却大大没有必要这么做，因为Glide本身就内置了一个ExternalCacheDiskCacheFactory，可以允许将加载的图片都缓存到SD卡。

那么接下来，我们就尝试使用这个ExternalCacheDiskCacheFactory来替换默认的InternalCacheDiskCacheFactory，从而将所有Glide加载的图片都缓存到SD卡上。

由于在前面我们已经创建好了一个自定义模块MyGlideModule，那么现在就可以直接在这里编写逻辑了，代码如下所示：

```
public class MyGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        builder.setDiskCache(new
    ExternalCacheDiskCacheFactory(context));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {

    }

}
```

没错，就是这么简单，现在所有Glide加载的图片都会缓存到SD卡上了。

另外，InternalCacheDiskCacheFactory和ExternalCacheDiskCacheFactory的默认硬盘缓存大小都是250M。也就是说，如果你的应用缓存的图片总大小超出了250M，那么Glide就会按照DiskLruCache算法的原则来清理缓存的图片。

当然，我们是可以对这个默认的缓存大小进行修改的，而且修改方式非常简单，如下所示：

```
public class MyGlideModule implements GlideModule {

    public static final int DISK_CACHE_SIZE = 500 * 1024
    * 1024;

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        builder.setDiskCache(new
ExternalCacheDiskCacheFactory(context, DISK_CACHE_SIZE));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {

}

}
```

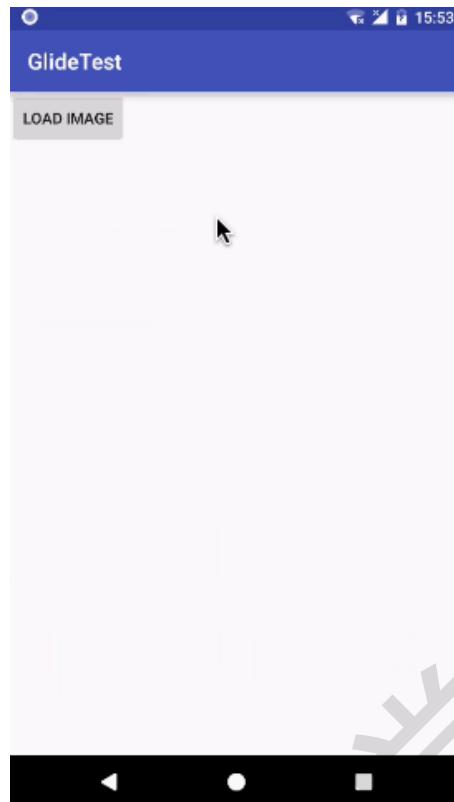
只需要向ExternalCacheDiskCacheFactory或者InternalCacheDiskCacheFactory再传入一个参数就可以了，现在我们就将Glide硬盘缓存的大小调整成了500M。

好了，更改Glide配置的功能就是这么简单，那么接下来我们就来验证一下更改的配置到底有没有生效吧。

这里还是使用最基本的Glide加载语句来去加载一张网络图片：

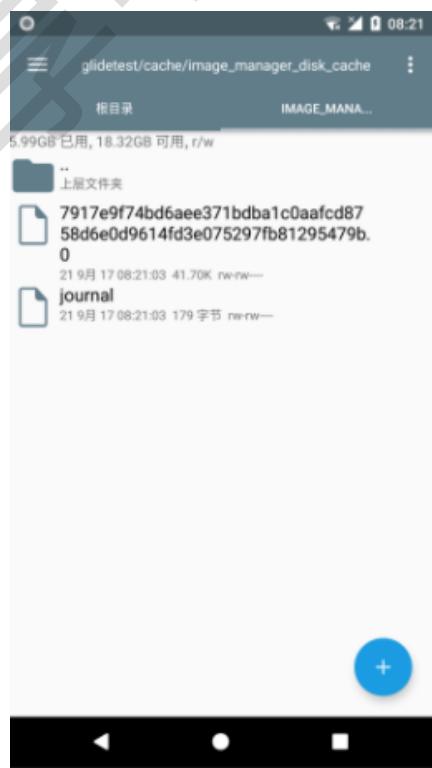
```
String url = "http://guolin.tech/book.png";
Glide.with(this)
    .load(url)
    .into(imageview);
```

运行一下程序，效果如下图所示：



OK，现在图片已经加载出现了，那么我们去找一找它的缓存吧。

ExternalCacheDiskCacheFactory的默认缓存路径是在sdcard/Android/包名/cache/image_manager_disk_cache目录当中，我们使用文件浏览器进入到这个目录，结果如下图所示。



可以看到，这里有两个文件，其中journal文件是DiskLruCache算法的日志文件，这个文件必不可少，且只会有一个。想了解更多关于DiskLruCache算法的朋友，可以去阅读我的这篇博客 [Android DiskLruCache完全解析，硬盘缓存的最佳方案](#)。

而另外一个文件就是那张缓存的图片了，它的文件名虽然看上去很奇怪，但是我们只需要把这个文件的后缀改成.png，然后用图片浏览器打开，结果就一目了然了，如下图所示。



由此证明，我们已经成功将Glide的硬盘缓存路径修改到SD卡上了。

另外这里再提一点，我们都知道Glide和Picasso的用法是非常相似的，但是有一点差别却很大。Glide加载图片的默认格式是RGB_565，而Picasso加载图片的默认格式是ARGB_8888。ARGB_8888格式的图片效果会更加细腻，但是内存开销会比较大。而RGB_565格式的图片则更加节省内存，但是图片效果上会差一些。

Glide和Picasso各自采取的默认图片格式谈不上孰优孰劣，只能说各自的取舍不一样。但是如果你希望Glide也能使用ARGB_8888的图片格式，这当然也是可以的。我们只需要在MyGlideModule中更改一下默认配置即可，如下所示：

```
public class MyGlideModule implements GlideModule {
```

```
public static final int DISK_CACHE_SIZE = 500 * 1024
* 1024;

@Override
public void applyOptions(Context context,
GlideBuilder builder) {
    builder.setDiskCache(new
ExternalCacheDiskCacheFactory(context, DISK_CACHE_SIZE));

builder.setDecodeFormat(DecodeFormat.PREFER_ARGB_8888);
}

@Override
public void registerComponents(Context context, Glide
glide) {

}

}
```

通过这样配置之后，使用Glide加载的所有图片都将会使用ARGB_8888的格式，虽然图片质量变好了，但同时内存开销也会明显增大，所以你要做好心理准备哦。

好了，关于更改Glide配置的内容就介绍这么多，接下来就让我们进入到下一个非常重要的主题，替换Glide组件。

替换Glide组件

替换Glide组件功能需要在自定义模块的registerComponents()方法中加入具体的替换逻辑。相比于更改Glide配置，替换Glide组件这个功能的难度就明显大了不少。Glide中的组件非常繁多，也非常复杂，但其实大多数情况下并不需要我们去做什么替换。不过，有一个组件却有着比较大的替换需求，那就是Glide的HTTP通讯组件。

默认情况下，Glide使用的是基于原生HttpURLConnection进行订制的HTTP通讯组件，但是现在大多数的Android开发者都更喜欢使用OkHttp，因此将Glide中的HTTP通讯组件修改成OkHttp的这个需求比较常见，那么今天我们也会以这个功能来作为例子进行讲解。

首先来看一下Glide中目前有哪些组件吧，在Glide类的构造方法当中，如下所示：

```
public class Glide {  
  
    Glide(Engine engine, MemoryCache memoryCache,  
    BitmapPool bitmapPool, Context context, DecodeFormat  
    decodeFormat) {  
        ...  
  
        register(File.class, ParcelFileDescriptor.class,  
        new FileDescriptorFileLoader.Factory());  
        register(File.class, InputStream.class, new  
        StreamFileLoader.Factory());  
        register(int.class, ParcelFileDescriptor.class,  
        new FileDescriptorResourceLoader.Factory());  
        register(int.class, InputStream.class, new  
        StreamResourceLoader.Factory());  
        register(Integer.class,  
        ParcelFileDescriptor.class, new  
        FileDescriptorResourceLoader.Factory());  
        register(Integer.class, InputStream.class, new  
        StreamResourceLoader.Factory());  
        register(String.class,  
        ParcelFileDescriptor.class, new  
        FileDescriptorStringLoader.Factory());  
        register(String.class, InputStream.class, new  
        StreamStringLoader.Factory());  
        register(Uri.class, ParcelFileDescriptor.class,  
        new FileDescriptorUriLoader.Factory());  
        register(Uri.class, InputStream.class, new  
        StreamUriLoader.Factory());
```

```
        register(URL.class, InputStream.class, new
StreamUrlLoader.Factory());
        register(GlideUrl.class, InputStream.class, new
HttpUrlGlideUrlLoader.Factory());
        register(byte[].class, InputStream.class, new
StreamByteArrayLoader.Factory());

        ...
}

}
```

可以看到，这里都是以调用register()方法的方式来注册一个组件，register()方法中传入的参数表示Glide支持使用哪种参数类型来加载图片，以及如何去处理这种类型的图片加载。举个例子：

```
register(GlideUrl.class, InputStream.class, new
HttpUrlGlideUrlLoader.Factory());
```

这句代码就表示，我们可以使用Glide.with(context).load(new GlideUrl("url...")).into(imageView)的方式来加载图片，而 HttpUrlGlideUrlLoader.Factory则是要负责处理具体的网络通讯逻辑。如果我们想要将Glide的HTTP通讯组件替换成OkHttp的话，那么只需要在自定义模块当中重新注册一个GlideUrl类型的组件就行了。

说到这里有的朋友可能会疑问了，我们平时使用Glide加载图片时，大多数情况下都是直接将图片的URL字符串传入到load()方法当中的，很少会将它封装成GlideUrl对象之后再传入到load()方法当中，那为什么只需要重新注册一个GlideUrl类型的组件，而不需要去重新注册一个String类型的组件呢？其实道理很简单，因为load(String)方法只是Glide给我们提供一种简易的API封装而已，它的底层仍然还是调用的GlideUrl组件，因此我们在替换组件的时候只需要直接替换最底层的，这样就一步到位了。

那么接下来我们就开始学习到底如何将Glide的HTTP通讯组件替换成OkHttp。

首先第一步，不用多说，肯定是要先将OkHttp的库引入到当前项目中，如下所示：

```
dependencies {
    compile 'com.squareup.okhttp3:okhttp:3.9.0'
}
```

然后接下来该怎么做呢？我们只要依葫芦画瓢就可以了。刚才不是说 Glide的网络通讯逻辑是由HttpUrlGlideUrlLoader.Factory来负责的吗，那么我们就来看一下它的源码：

```
public class HttpUrlGlideUrlLoader implements
ModelLoader<GlideUrl, InputStream> {

    private final ModelCache<GlideUrl, GlideUrl>
modelCache;

    public static class Factory implements
ModelLoaderFactory<GlideUrl, InputStream> {
        private final ModelCache<GlideUrl, GlideUrl>
modelCache = new ModelCache<GlideUrl, GlideUrl>(500);

        @Override
        public ModelLoader<GlideUrl, InputStream>
build(Context context, GenericLoaderFactory factories) {
            return new HttpUrlGlideUrlLoader(modelCache);
        }

        @Override
        public void teardown() {
        }
    }

    public HttpUrlGlideUrlLoader() {
        this(null);
    }

    public HttpUrlGlideUrlLoader(ModelCache<GlideUrl,
GlideUrl> modelCache) {
        this.modelCache = modelCache;
    }
}
```

```
    @Override
    public DataFetcher<InputStream>
getResourceFetcher(GlideUrl model, int width, int height)
{
    GlideUrl url = model;
    if (modelCache != null) {
        url = modelCache.get(model, 0, 0);
        if (url == null) {
            modelCache.put(model, 0, 0, model);
            url = model;
        }
    }
    return new HttpUrlFetcher(url);
}
}
```

可以看到，`HttpUrlGlideUrlLoader.Factory`是一个内部类，外层的`HttpUrlGlideUrlLoader`类实现了`ModelLoader<GlideUrl, InputStream>`这个接口，并重写了`getResourceFetcher()`方法。而在`getResourceFetcher()`方法中，又创建了一个`HttpUrlFetcher`的实例，在这里才是真正处理具体网络通讯逻辑的地方，代码如下所示：

```
public class HttpUrlFetcher implements
DataFetcher<InputStream> {
    private static final String TAG = "HttpUrlFetcher";
    private static final int MAXIMUM_REDIRECTS = 5;
    private static final HttpUrlConnectionFactory
DEFAULT_CONNECTION_FACTORY = new
DefaultHttpUrlConnectionFactory();

    private final GlideUrl glideUrl;
    private final HttpUrlConnectionFactory
connectionFactory;

    private HttpURLConnection urlConnection;
    private InputStream stream;
```

```
private volatile boolean isCancelled;

public HttpURLConnectionFetcher(GlideUrl glideUrl) {
    this(glideUrl, DEFAULT_CONNECTION_FACTORY);
}

HttpURLConnectionFetcher(GlideUrl glideUrl,
HttpURLConnectionFactory connectionFactory) {
    this.glideUrl = glideUrl;
    this.connectionFactory = connectionFactory;
}

@Override
public InputStream loadData(Priority priority) throws
Exception {
    return loadDataWithRedirects(glideUrl.toURL(), 0
, null , glideUrl.getHeaders());
}

private InputStream loadDataWithRedirects(URL url,
int redirects, URL lastUrl, Map<String, String> headers)
throws IOException {
    if (redirects >= MAXIMUM_REDIRECTS) {
        throw new IOException("Too many (> " +
MAXIMUM_REDIRECTS + ") redirects!");
    } else {
        try {
            if (lastUrl != null &&
url.toURI().equals(lastUrl.toURI())) {
                throw new IOException("In re-direct
loop");
            }
        } catch (URISyntaxException e) {
        }
    }
    urlConnection = connectionFactory.build(url);
    for (Map.Entry<String, String> headerEntry :
headers.entrySet()) {
```

```
urlConnection.addRequestProperty(headerEntry.getKey() ,  
headerEntry.getValue());  
}  
urlConnection.setConnectTimeout(2500);  
urlConnection.setReadTimeout(2500);  
urlConnection.setUseCaches(false);  
urlConnection.connect();  
if (isCancelled) {  
    return null;  
}  
final int statusCode =  
urlConnection.getResponseCode();  
if (statusCode / 100 == 2) {  
    return  
getStreamForSuccessfulRequest(urlConnection);  
} else if (statusCode / 100 == 3) {  
    String redirect urlString =  
urlConnection.getHeaderField("Location");  
    if (TextUtils.isEmpty(redirect urlString)) {  
        throw new IOException("Received empty or  
null redirect url");  
    }  
    URL redirectUrl = new URL(url,  
redirect urlString);  
    return loadDataWithRedirects(redirectUrl,  
redirects + 1, url, headers);  
} else {  
    if (statusCode == -1) {  
        throw new IOException("Unable to retrieve  
response code from HttpURLConnection.");  
    }  
    throw new IOException("Request failed " +  
statusCode + ": " + urlConnection.getResponseMessage());  
}  
}
```

```
    private InputStream
getStreamForSuccessfulRequest(HttpURLConnection
URLConnection)
        throws IOException {
    if
(TextUtils.isEmpty(URLConnection.getContentEncoding())) {
        int contentLength =
URLConnection.getContentLength();
        stream =
ContentLengthInputStream.obtain(URLConnection.getInputStream(),
contentLength);
    } else {
        stream = URLConnection.getInputStream();
    }
    return stream;
}

@Override
public void cleanup() {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException e) {
        }
    }
    if (URLConnection != null) {
        URLConnection.disconnect();
    }
}

@Override
public String getId() {
    return glideUrl.getCacheKey();
}

@Override
public void cancel() {
    isCancelled = true;
}
```

```
}

interface HttpURLConnectionFactory {
    HttpURLConnection build(URL url) throws
IOException;
}

private static class DefaultHttpURLConnectionFactory
implements HttpURLConnectionFactory {
    @Override
    public HttpURLConnection build(URL url) throws
IOException {
        return (HttpURLConnection)
url.openConnection();
    }
}
```

上面这段代码看上去应该不费力吧？其实就是一些HttpURLConnection的用法而已。那么我们只需要仿照着HttpUrlFetcher的代码来写，并且把HTTP的通讯组件替换成OkHttp就可以了。

现在新建一个OkHttpFetcher类，并且同样实现DataFetcher接口，代码如下所示：

```
public class OkHttpFetcher implements
DataFetcher<InputStream> {

    private final OkHttpClient client;
    private final GlideUrl url;
    private InputStream stream;
    private ResponseBody responseBody;
    private volatile boolean isCancelled;

    public OkHttpFetcher(OkHttpClient client, Glideurl
url) {
        this.client = client;
```

```
        this.url = url;
    }

    @Override
    public InputStream loadData(Priority priority) throws
Exception {
    Request.Builder requestBuilder = new
Request.Builder()
        .url(url.toStringUrl());
    for (Map.Entry<String, String> headerEntry :
url.getHeaders().entrySet()) {
        String key = headerEntry.getKey();
        requestBuilder.addHeader(key,
headerEntry.getValue());
    }
    requestBuilder.addHeader("httpplib", "okhttp");
    Request request = requestBuilder.build();
    if (isCancelled) {
        return null;
    }
    Response response =
client.newCall(request).execute();
    responseBody = response.body();
    if (!response.isSuccessful() || responseBody ==
null) {
        throw new IOException("Request failed with
code: " + response.code());
    }
    stream =
ContentLengthInputStream.obtain(responseBody.byteStream()
,
                    responseBody.contentLength());
    return stream;
}

@Override
public void cleanup() {
    try {
```

```
        if (stream != null) {
            stream.close();
        }
        if (responseBody != null) {
            responseBody.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
public String getId() {
    return url.getCacheKey();
}

@Override
public void cancel() {
    isCancelled = true;
}
}
```

上面这段代码完全就是我照着HttpUrlFetcher依葫芦画瓢写出来的，用的也都是一些OkHttp的基本用法，相信不需要再做什么解释了吧。可以看到，使用OkHttp来编写网络通讯的代码要比使用HttpURLConnection简单很多，代码行数也少了很多。注意在第22行，我添加了一个httpplib: OkHttp的请求头，这个是待会儿我们用来进行测试验证的，大家实际项目中的代码无须添加这个请求头。

那么我们就继续发挥依葫芦画瓢的精神，仿照着HttpUrlGlideUrlLoader 再写一个OkHttpGlideUrlLoader吧。新建一个OkHttpGlideUrlLoader类，并且实现ModelLoader<.GlideUrl, InputStream>接口，代码如下所示：

```
public class okHttpGlideurlLoader implements
ModelLoader<.GlideUrl, InputStream> {
```

```
private OkHttpClient okHttpClient;

public static class Factory implements
ModelLoaderFactory<.GlideUrl, InputStream> {

    private OkHttpClient client;

    public Factory() {
    }

    public Factory(OkHttpClient client) {
        this.client = client;
    }

    private synchronized OkHttpClient
getOkHttpClient() {
        if (client == null) {
            client = new OkHttpClient();
        }
        return client;
    }

    @Override
    public ModelLoader< GlideUrl, InputStream>
build(Context context, GenericLoaderFactory factories) {
        return new
OkHttpGlideUrlLoader(getOkHttpClient());
    }

    @Override
    public void teardown() {
    }

    public OkHttpGlideUrlLoader(OkHttpClient client) {
        this.okHttpClient = client;
    }
}
```

```
    @Override  
    public DataFetcher<InputStream>  
getResourceFetcher(GlideUrl model, int width, int height)  
{  
    return new OkHttpFetcher(okHttpClient, model);  
}  
}
```

注意这里的Factory我提供了两个构造方法，一个是不带任何参数的，一个带OkHttpClient参数的。如果对OkHttp不需要进行任何自定义的配置，那么就调用无参的Factory构造函数即可，这样会在内部自动创建一个OkHttpClient实例。但如果你需要想添加拦截器，或者修改OkHttp的默认超时等等配置，那么就自己创建一个OkHttpClient的实例，然后传入到Factory的构造方法当中就行了。

好了，现在就只差最后一步，将我们刚刚创建的OkHttpGlideUrlLoader和OkHttpFetcher注册到Glide当中，将原来的HTTP通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {  
    ...  
    @Override  
    public void registerComponents(Context context, Glide  
glide) {  
        glide.register(GlideUrl.class, InputStream.class,  
new OkHttpGlideUrlLoader.Factory());  
    }  
}
```

可以看到，这里也是调用了Glide的register()方法来注册组件的。register()方法中使用的Map类型来存储已注册的组件，因此我们这里重新注册了一遍GlideUrl.class类型的组件，就把原来的组件给替换掉了。

理论上来说，现在我们已经成功将Glide的HTTP通讯组件替换成OkHttp了，现在唯一的问题就是我们该如何去验证一下到底有没有替换成功呢？

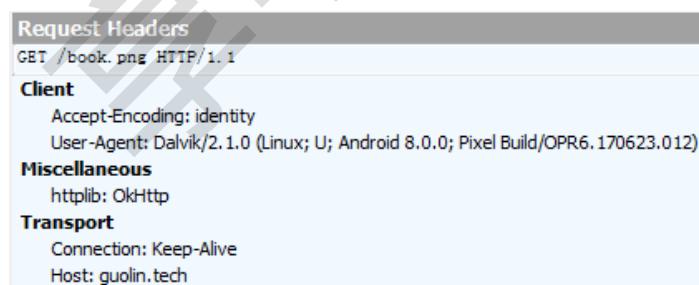
验证的方式我倒是想了很多种，比如添加OkHttp拦截器，或者自己架设一个测试用的服务器都是可以的。不过为了让大家最直接地看到验证结果，这里我准备使用Fiddler这个抓包工具来进行验证。这个工具的用法非常简单，但是限于篇幅我就不在本篇文章中介绍这个工具的用法了，还没用过这个工具的朋友们可以通过[这篇文章](#)了解一下。

在开始验证之前，我们还得要再修改一下Glide加载图片的代码才行，如下所示：

```
String url = "http://guolin.tech/book.png";
Glide.with(this)
    .load(url)
    .skipMemoryCache(true)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

这里我把Glide的内存缓存和硬盘缓存都禁用掉了，不然的话，Glide可能会直接读取刚才缓存的图片，而不会再重新发起网络请求。

好的，现在我们重新使用Glide加载一下图片，然后观察Fiddler中的抓包情况，如下图所示。



可以看到，在HTTP请求头中确实有我们刚才自己添加的`httpplib: OkHttp`。也就说明，Glide的HTTP通讯组件的确被替换成功了。

更简单的组件替换

上述方法是我们纯手工地将Glide的HTTP通讯组件进行了替换，如果你不想这么麻烦也是可以的，Glide官方给我们提供了非常简便的HTTP组件替换方式。并且除了支持OkHttp3之外，还支持OkHttp2和Volley。

我们只需要在gradle当中添加几行库的配置就行了。比如使用OkHttp3来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.squareup.okhttp3:okhttp:3.9.0'  
    compile 'com.github.bumptech.glide:okhttp3-  
integration:1.5.0@aar'  
}
```

使用OkHttp2来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.github.bumptech.glide:okhttp-  
integration:1.5.0@aar'  
    compile 'com.squareup.okhttp:okhttp:2.7.5'  
}
```

使用Volley来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.github.bumptech.glide:volley-  
integration:1.5.0@aar'  
    compile 'com.mcxiaoke.volley:library:1.0.19'  
}
```

当然了，这些库背后的工作原理和我们刚才自己手动实现替换HTTP组件的原理是一模一样的。而学会了手动替换组件的原理我们就能更加轻松地扩展更多丰富的功能，因此掌握这一技能还是非常重要的。

好了，那么今天的文章就到这里了。

3.8 Glide源码学习八：实现带进度的Glide图片加载功能

扩展目标

首先来确立一下功能扩展的目标。虽说Glide本身就已经十分强大了，但是有一个功能却长期以来都不支持，那就是监听下载进度功能。

我们都知道，使用Glide来加载一张网络上的图片是非常简单的，但是让人头疼的是，我们却无从得知当前图片的下载进度。如果这张图片很小的话，那么问题也不大，反正很快就会被加载出来。但如果这是一张比较大的GIF图，用户耐心等了很久结果图片还没显示出来，这个时候你就会觉得下载进度功能是十分有必要的了。

好的，那么我们今天的目标就是对Glide进行功能扩展，使其支持监听图片下载进度的功能。

开始

今天这篇文章我会带着大家从零去创建一个新的项目，一步步地进行实现，最终完成一个带进度的Glide图片加载的Demo。当然，在本篇文章的最后我会提供这个Demo的完整源码，但是这里我仍然希望大家能用心跟着我一步步来编写。

那么我们现在就开始吧，首先创建一个新项目，就叫做GlideProgressTest吧。

项目创建完成后的第一件事就是要将必要的依赖库引入到当前的项目当中，目前我们必须要依赖的两个库就是Glide和OkHttp。在app/build.gradle文件当中添加如下配置：

```
dependencies {  
    compile 'com.github.bumptech.glide:glide:3.7.0'  
    compile 'com.squareup.okhttp3:okhttp:3.9.0'  
}
```

另外，由于Glide和OkHttp都需要用到网络功能，因此我们还得在AndroidManifest.xml中声明一下网络权限才行：

```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

好了，这样准备工作就完成了。

替换通讯组件

通过第二篇文章的源码分析，我们知道了Glide内部HTTP通讯组件的底层实现是基于HttpURLConnection来进行定制的。但是HttpURLConnection的可扩展性比较有限，我们在它的基础之上无法实现监听下载进度的功能，因此今天的第一大动作就是要将Glide中的HTTP通讯组件替换成OkHttp。

关于HTTP通讯组件的替换原理和替换方式，我在第六篇文章当中都介绍得比较清楚了，这里就不再赘述。下面我们就来开始快速地替换一下。

新建一个OkHttpFetcher类，并且实现DataFetcher接口，代码如下所示：

```
public class OkHttpFetcher implements DataFetcher<InputStream> {

    private final OkHttpClient client;
    private final GlideUrl url;
    private InputStream stream;
    private ResponseBody responseBody;
    private volatile boolean isCancelled;

    public OkHttpFetcher(OkHttpClient client, GlideUrl url) {
        this.client = client;
        this.url = url;
    }

    @Override
    public InputStream loadData(Priority priority) throws Exception {
        Request.Builder requestBuilder = new Request.Builder()
                .url(url.toStringUrl());
        for (Map.Entry<String, String> headerEntry : url.getHeaders().entrySet()) {
            String key = headerEntry.getKey();
            requestBuilder.addHeader(key,
                    headerEntry.getValue());
        }
        return client.newCall(requestBuilder.build()).execute().body().byteStream();
    }

    @Override
    public void cancel() {
        isCancelled = true;
    }

    @Override
    public void cleanup() {
        if (stream != null) {
            try {
                stream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }

    Request request = requestBuilder.build();
    if (isCancelled) {
        return null;
    }

    Response response =
client.newCall(request).execute();
    responseBody = response.body();
    if (!response.isSuccessful() || responseBody ==
null) {
        throw new IOException("Request failed with
code: " + response.code());
    }

    stream =
ContentLengthInputStream.obtain(responseBody.byteStream()
,
                    responseBody.contentLength());
    return stream;
}

@Override
public void cleanup() {
    try {
        if (stream != null) {
            stream.close();
        }
        if (responseBody != null) {
            responseBody.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
public String getId() {
    return url.getCacheKey();
}
```

```
    @Override  
    public void cancel() {  
        isCancelled = true;  
    }  
}
```

然后新建一个OkHttpGlideUrlLoader类，并且实现ModelLoader

```
public class OkHttpGlideUrlLoader implements  
ModelLoader<GlideUrl, InputStream> {  
  
    private OkHttpClient okHttpClient;  
  
    public static class Factory implements  
ModelLoaderFactory<GlideUrl, InputStream> {  
  
        private OkHttpClient client;  
  
        public Factory() {  
        }  
  
        public Factory(OkHttpClient client) {  
            this.client = client;  
        }  
  
        private synchronized OkHttpClient  
getOkHttpClient() {  
            if (client == null) {  
                client = new OkHttpClient();  
            }  
            return client;  
        }  
  
        @Override  
        public ModelLoader<GlideUrl, InputStream>  
build(Context context, GenericLoaderFactory factories) {
```

```
        return new
OkHttpGlideUrlLoader(getOkHttpClient());
    }

    @Override
    public void teardown() {
    }

}

public OkHttpGlideUrlLoader(OkHttpClient client) {
    this.okHttpClient = client;
}

@Override
public DataFetcher<InputStream>
getResourceFetcher(GlideUrl model, int width, int height)
{
    return new OkHttpFetcher(okHttpClient, model);
}
}
```

接下来，新建一个MyGlideModule类并实现GlideModule接口，然后在registerComponents()方法中将我们刚刚创建的OkHttpGlideUrlLoader和OkHttpFetcher注册到Glide当中，将原来的HTTP通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
        glide.register(GlideUrl.class, InputStream.class,
new OkHttpGlideUrlLoader.Factory());
    }
}
```

最后，为了让Glide能够识别我们自定义的MyGlideModule，还得在AndroidManifest.xml文件当中加入如下配置才行：

```
<manifest>
    ...
    <application>
        <meta-data
            android:name="com.example.glideprogressstest.MyGlideModule"
            android:value="GlideModule" />
        ...
    </application>
</manifest>
```

OK，这样我们就把Glide中的HTTP通讯组件成功替换成OkHttp了。

实现下载进度监听

那么，将HTTP通讯组件替换成OkHttp之后，我们又该如何去实现监听下载进度的功能呢？这就要依靠OkHttp强大的拦截器机制了。

我们只要向OkHttp中添加一个自定义的拦截器，就可以在拦截器中捕获到整个HTTP的通讯过程，然后加入一些自己的逻辑来计算下载进度，这样就可以实现下载进度监听的功能了。

拦截器属于OkHttp的高级功能，不过即使你之前并没有接触过拦截器，我相信你也能轻松看懂本篇文章的，因为它本身并不难。

确定了实现思路之后，那我们就开始动手吧。首先创建一个没有任何逻辑的空拦截器，新建ProgressInterceptor类并实现Interceptor接口，代码如下所示：

```
public class ProgressInterceptor implements Interceptor {

    @Override
    public Response intercept(Chain chain) throws
    IOException {
        Request request = chain.request();
        Response response = chain.proceed(request);
        return response;
    }

}
```

这个拦截器中我们可以说是什么都没有做。就是拦截到了OkHttp的请求，然后调用proceed()方法去处理这个请求，最终将服务器响应的Response返回。

接下来我们需要启用这个拦截器，修改MyGlideModule中的代码，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
        OkHttpClient.Builder builder = new
OkHttpClient.Builder();
        builder.addInterceptor(new
ProgressInterceptor());
        OkHttpClient okHttpClient = builder.build();
        glide.register(GlideUrl.class, InputStream.class,
new OkHttpGlideUrlLoader.Factory(okHttpClient));
    }
}
```

这里我们创建了一个OkHttpClient.Builder，然后调用addInterceptor()方法将刚才创建的ProgressInterceptor添加进去，最后将构建出来的新 OkHttpClient对象传入到OkHttpGlideUrlLoader.Factory中即可。

好的，现在自定义的拦截器已经启用了，接下来就可以开始去实现下载进度监听的具体逻辑了。首先新建一个ProgressListener接口，用于作为进度监听回调的工具，如下所示：

```
public interface ProgressListener {
    void onProgress(int progress);
}
```

然后我们在ProgressInterceptor中加入注册下载监听和取消注册下载监听的方法。修改ProgressInterceptor中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {
```

```
static final Map<String, ProgressListener>
LISTENER_MAP = new HashMap<>();

public static void addListener(String url,
ProgressListener listener) {
    LISTENER_MAP.put(url, listener);
}

public static void removeListener(String url) {
    LISTENER_MAP.remove(url);
}

@Override
public Response intercept(Chain chain) throws
IOException {
    Request request = chain.request();
    Response response = chain.proceed(request);
    return response;
}

}
```

可以看到，这里使用了一个Map来保存注册的监听器，Map的键是一个URL地址。之所以要这么做，是因为你可能会使用Glide同时加载很多张图片，而这种情况下，必须要能区分出来每个下载进度的回调到底是对应哪个图片URL地址的。

接下来就要到今天最复杂的部分了，也就是下载进度的具体计算。我们需要新建一个ProgressResponseBody类，并让它继承自OkHttp的ResponseBody，然后在这个类当中去编写具体的监听下载进度的逻辑，代码如下所示：

```
public class ProgressResponseBody extendsResponseBody {

    private static final String TAG =
"ProgressResponseBody";

    private BufferedSource bufferedSource;
```

```
private ResponseBody responseBody;

private ProgressListener listener;

public ProgressResponseBody(String url, ResponseBody responseBody) {
    this.responseBody = responseBody;
    listener =
ProgressInterceptor.LISTENER_MAP.get(url);
}

@Override
public MediaType contentType() {
    return responseBody.contentType();
}

@Override
public long contentLength() {
    return responseBody.contentLength();
}

@Override
public BufferedSource source() {
    if (bufferedSource == null) {
        bufferedSource = Okio.buffer(new
ProgressSource(responseBody.source()));
    }
    return bufferedSource;
}

private class ProgressSource extends ForwardingSource
{

    long totalBytesRead = 0;

    int currentProgress;
```

```
ProgressSource(Source source) {
    super(source);
}

@Override
public long read(Buffer sink, long byteCount)
throws IOException {
    long bytesRead = super.read(sink, byteCount);
    long fullLength =
responseBody.contentLength();
    if (bytesRead == -1) {
        totalBytesRead = fullLength;
    } else {
        totalBytesRead += bytesRead;
    }
    int progress = (int) (100f * totalBytesRead /
fullLength);
    Log.d(TAG, "download progress is " +
progress);
    if (listener != null && progress !=
currentProgress) {
        listener.onProgress(progress);
    }
    if (listener != null && totalBytesRead ==
fullLength) {
        listener = null;
    }
    currentProgress = progress;
    return bytesRead;
}
}
```

其实这段代码也不是很难，下面我来简单解释一下。首先，我们定义了一个ProgressResponseBody的构造方法，该构造方法中要求传入一个url参数和一个ResponseBody参数。那么很显然，url参数就是图片的url地址了，而ResponseBody参数则是OkHttp拦截到的原始的ResponseBody对象。然后在构造方法中，我们调用了ProgressInterceptor中的LISTENER_MAP来去获取该url对应的监听器回调对象，有了这个对象，待会就可以回调计算出来的下载进度了。

由于继承了ResponseBody类之后一定要重写contentType()、contentLength()和source()这三个方法，我们在contentType()和contentLength()方法中直接就调用传入的原始ResponseBody的contentType()和contentLength()方法即可，这相当于一种委托模式。但是在source()方法中，我们就必须加入点自己的逻辑了，因为这里要涉及到具体的下载进度计算。

那么我们具体看一下source()方法，这里先是调用了原始ResponseBody的source()方法来去获取Source对象，接下来将这个Source对象封装到了一个ProgressSource对象当中，最终再用Okio的buffer()方法封装成BufferedSource对象返回。

那么这个ProgressSource是什么呢？它是一个我们自定义的继承自ForwardingSource的实现类。ForwardingSource也是一个使用委托模式的工具，它不处理任何具体的逻辑，只是负责将传入的原始Source对象进行中转。但是，我们使用ProgressSource继承自ForwardingSource，那么就可以在中转的过程中加入自己的逻辑了。

可以看到，在ProgressSource中我们重写了read()方法，然后在read()方法中获取该次读取到的字节数以及下载文件的总字节数，并进行一些简单的数学计算就能算出当前的下载进度了。这里我先使用Log工具将算出的结果打印了一下，再通过前面获取到的回调监听器对象将结果进行回调。

好的，现在计算下载进度的逻辑已经完成了，那么我们快点在拦截器当中使用它吧。修改ProgressInterceptor中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {
```

```
    ...
```

```
    @Override
    public Response intercept(Chain chain) throws
    IOException {
        Request request = chain.request();
        Response response = chain.proceed(request);
        String url = request.url().toString();
       ResponseBody body = response.body();
        Response newResponse =
        response.newBuilder().body(new ProgressResponseBody(url,
        body)).build();
        return newResponse;
    }

}
```

这里也都是一些OkHttp的简单用法。我们通过Response的newBuilder()方法来创建一个新的Response对象，并把它的body替换成刚才实现的ProgressResponseBody，最终将新的Response对象进行返回，这样计算下载进度的逻辑就能生效了。

代码写到这里，我们就可以来运行一下程序了。现在无论是加载任何网络上的图片，都应该是可以监听到它的下载进度的。

修改activity_main.xml中的代码，如下所示：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Image"
        android:onClick="loadImage"
```

```
    />

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

很简单，这里使用了一个Button按钮来加载图片，使用了一个ImageView来展示图片。

然后修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

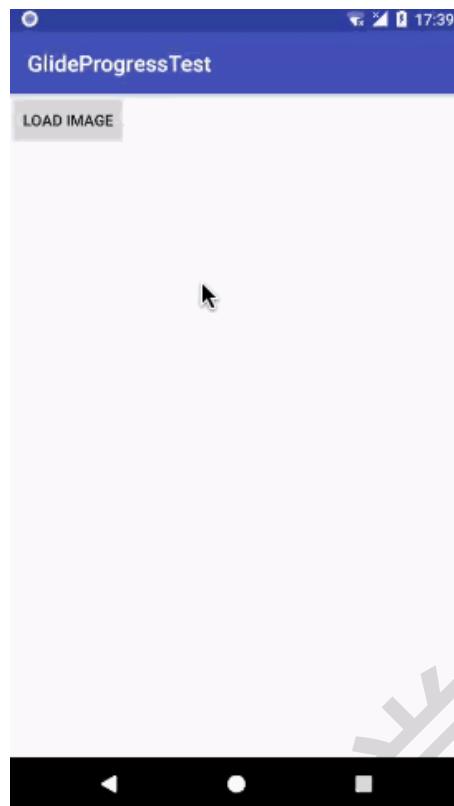
    String url = "http://guolin.tech/book.png";

    ImageView image;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.image);
    }

    public void loadImage(View view) {
        Glide.with(this)
            .load(url)
            .diskCacheStrategy(DiskCacheStrategy.NONE)
            .override(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL)
            .into(image);
    }
}
```

现在就可以运行一下程序了，效果如下图所示。



OK，图片已经加载出来了。那么怎么验证有没有成功监听到图片的下载进度呢？还记得我们刚才在ProgressResponseBody中加的打印日志吗？现在只要去logcat中观察一下就知道了，如下图所示：

```
D/ProgressResponseBody: download progress is 0
D/ProgressResponseBody: download progress is 6
D/ProgressResponseBody: download progress is 8
D/ProgressResponseBody: download progress is 9
D/ProgressResponseBody: download progress is 10
D/ProgressResponseBody: download progress is 11
D/ProgressResponseBody: download progress is 12
D/ProgressResponseBody: download progress is 13
D/ProgressResponseBody: download progress is 14
D/ProgressResponseBody: download progress is 15
D/ProgressResponseBody: download progress is 16
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 18
D/ProgressResponseBody: download progress is 19
D/ProgressResponseBody: download progress is 20
D/ProgressResponseBody: download progress is 21
D/ProgressResponseBody: download progress is 22
D/ProgressResponseBody: download progress is 23
D/ProgressResponseBody: download progress is 24
D/ProgressResponseBody: download progress is 25
D/ProgressResponseBody: download progress is 26
D/ProgressResponseBody: download progress is 27
```

由此可见，下载进度监听功能已经成功实现了。

进度显示

虽然现在我们已经能够监听到图片的下载进度了，但是这个进度目前还只能显示在控制台打印当中，这对于用户来说是没有任何意义的，因此我们下一步就是要想办法将下载进度显示到界面上。

现在修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    String url = "http://guolin.tech/book.png";

    ImageView image;

    ProgressDialog progressDialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.image);
        progressDialog = new ProgressDialog(this);

        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progressDialog.setMessage("加载中");
    }

    public void loadImage(View view) {
        ProgressInterceptor.addListener(url, new
ProgressListener() {
            @Override
            public void onProgress(int progress) {
                progressDialog.setProgress(progress);
            }
        });
        Glide.with(this)
            .load(url)
            .diskCacheStrategy(DiskCacheStrategy.NONE)
```

```
        .override(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL)
        .into(new
GlideDrawableImageViewTarget(image) {
    @Override
    public void onLoadStarted(Drawable
placeholder) {
        super.onLoadStarted(placeholder);
        progressDialog.show();
    }

    @Override
    public void
onResourceReady(GlideDrawable resource, GlideAnimation<?
super GlideDrawable> animation) {
        super.onResourceReady(resource,
animation);
        progressDialog.dismiss();

ProgressInterceptor.removeListener(url);
    }
});
}
```

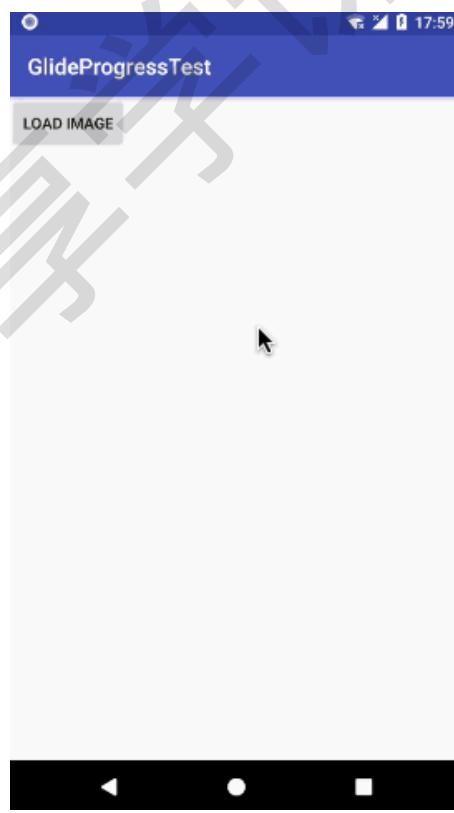
代码并不复杂。这里我们新增了一个ProgressDialog用来显示下载进度，然后在loadImage()方法中，调用了ProgressInterceptor.addListener()方法来去注册一个下载监听器，并在onProgress()回调方法中更新当前的下载进度。

最后，Glide的into()方法也做了修改，这次是into到了一个GlideDrawableImageViewTarget当中。我们重写了它的onLoadStarted()方法和onResourceReady()方法，从而实现当图片开始加载的时候显示进度对话框，当图片加载完成时关闭进度对话框的功能。

现在重新运行一下程序，效果如下图所示。



当然，不仅仅是静态图片，体积比较大的GIF图也是可以成功监听到下载进度的。比如我们把图片的url地址换成<http://guolin.tech/test.gif>，重新运行程序，效果如下图所示。



好了，这样我们就把带进度的Glide图片加载功能完整地实现了一遍。虽然这个例子当中的界面都比较粗糙，下载进度框也是使用的最简陋的，不过只要将功能学会了，界面那都不是事，大家后期可以自己进行各种界面优化。

最后，如果你想要下载完整的Demo，请[点击这里](#)。

3.9 Glide源码学习九：带你全面了解Glide 4的用法

其实在写这个系列第一篇文章的时候，Glide就推出4.0.0的RC版了。那个时候因为我一直研究的都是Glide 3.7.0版本，再加上RC版本还不太稳定，因此整个系列也都是基于3.7.0版本来写的。

而现在，Glide的最新版本已经出到了4.4.0，可以说Glide 4已经是相当成熟和稳定了。而且也不断有朋友一直在留言，想让我讲一讲Glide 4的用法，因为Glide 4相对于Glide 3改动貌似还是挺大的，学完了Glide 3再去使用Glide 4，发现根本就无法使用。

OK，那么今天就让我们用《带你全面了解Glide 4的用法》这样一篇文章，给这个Glide系列画上一个圆满的句号。

Glide 4概述

刚才有说到，有些朋友觉得Glide 4相对于Glide 3改动非常大，其实不然。之所以大家会有这种错觉，是因为你将Glide 3的用法直接搬到Glide 4中去使用，结果IDE全面报错，然后大家可能就觉得Glide 4的用法完全变掉了。

其实Glide 4相对于Glide 3的变动并不大，只是你还没有了解它的变动规则而已。一旦你掌握了Glide 4的变动规则之后，你会发现大多数Glide 3的用法放到Glide 4上都还是通用的。

我对Glide 4进行了一个大概的研究之后，发现Glide 4并不能算是有什么突破性的升级，而更多是一些API工整方面的优化。相比于Glide 3的API，Glide 4进行了更加科学合理地调整，使得易读性、易写性、可扩展性等方面都有了不错的提升。但如果你已经对Glide 3非常熟悉的话，并不

是就必须要切换到Glide 4上来，因为Glide 4上能实现的功能Glide 3也都能实现，而且Glide 4在性能方面也并没有什么提升。

但是对于新接触Glide的朋友而言，那就没必要再去学习Glide 3了，直接上手Glide 4就是最佳的选择了。

好了，对Glide 4进行一个基本的概述之后，接下来我们就要正式开始学习它的用法了。刚才我已经说了，Glide 4的用法相对于Glide 3其实改动并不大。在前面的七篇文章中，我们已经学习了Glide 3的基本用法、缓存机制、回调与监听、图片变换、自定义模块等用法，那么今天这篇文章的目标就很简单了，就是要掌握如何在Glide 4上实现之前所学习过的所有功能，那么我们现在就开始吧。

开始

要想使用Glide，首先需要将这个库引入到我们的项目当中。新建一个Glide4Test项目，然后在app/build.gradle文件当中添加如下依赖：

```
dependencies {
    implementation
    'com.github.bumptech.glide:glide:4.4.0'
    annotationProcessor
    'com.github.bumptech.glide:compiler:4.4.0'
}
```

注意，相比于Glide 3，这里要多添加一个compiler的库，这个库是用于生成Generated API的，待会我们会讲到它。

另外，Glide中需要用到网络功能，因此你还得在AndroidManifest.xml中声明一下网络权限才行：

就这么简单，然后我们就可以自由地使用Glide中的任意功能了。

加载图片

现在我们就来尝试一下如何使用Glide来加载图片吧。比如这是一张图片的地址：

```
http://guolin.tech/book.png
```

然后我们想要在程序当中去加载这张图片。

那么首先打开项目的布局文件，在布局当中加入一个Button和一个 ImageView，如下所示：

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:orientation="vertical">  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Load Image"  
        android:onClick="loadImage"  
    />  
  
    <ImageView  
        android:id="@+id/image_view"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent" />  
  
</LinearLayout>
```

为了让用户点击Button的时候能够将刚才的图片显示在ImageView上，我们需要修改MainActivity中的代码，如下所示：

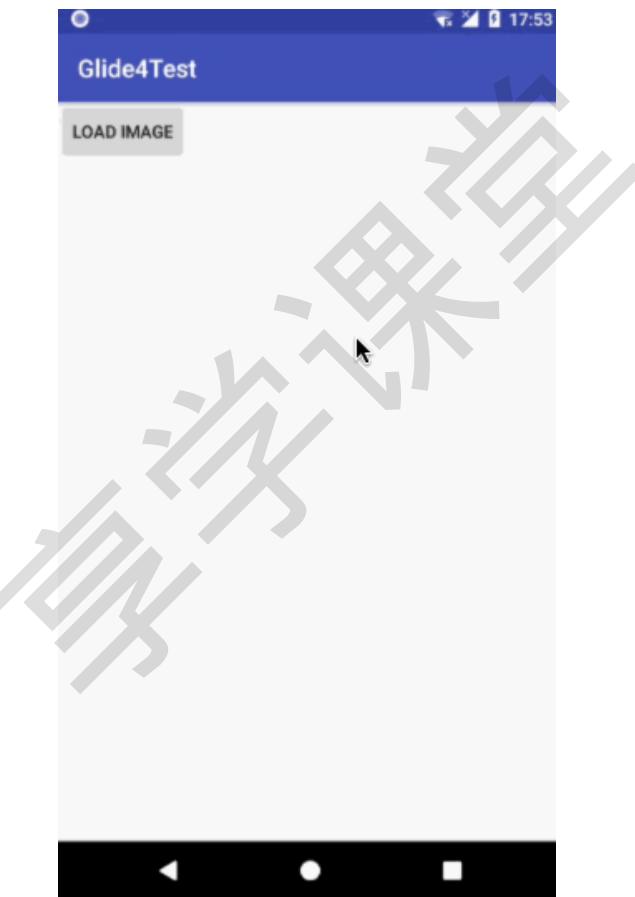
```
public class MainActivity extends AppCompatActivity {  
  
    ImageView imageView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);
```

```
    imageView = (ImageView)
findViewById(R.id.image_view);
}

public void loadImage(View view) {
    String url = "http://guolin.tech/book.png";
    Glide.with(this).load(url).into(imageView);
}

}
```

没错，就是这么简单。现在我们来运行一下程序，效果如下图所示：



可以看到，一张网络上的图片已经被成功下载，并且展示到ImageView上了。

你会发现，到目前为止，Glide 4的用法和Glide 3是完全一样的，实际上核心的代码就只有这一行而已：

```
Glide.with(this).load(url).into(imageView);
```

占位图

观察刚才加载网络图片的效果，你会发现，点击了Load Image按钮之后，要稍微等一会图片才会显示出来。这其实很容易理解，因为从网络上下载图片本来就是需要时间的。那么我们有没有办法再优化一下用户体验呢？当然可以，Glide提供了各种各样非常丰富的API支持，其中就包括了占位图功能。

顾名思义，占位图就是指在图片的加载过程中，我们先显示一张临时的图片，等图片加载出来了再替换成要加载的图片。

下面我们就来学习一下Glide占位图功能的使用方法，首先我事先准备好了张loading.jpg图片，用来作为占位图显示。然后修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.loading);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

没错，就是这么简单。这里我们先创建了一个RequestOptions对象，然后调用它的placeholder()方法来指定占位图，再将占位图片的资源id传入到这个方法中。最后，在Glide的三步走之间加入一个apply()方法，来应用我们刚才创建的RequestOptions对象。

不过如果你现在重新运行一下代码并点击Load Image，很可能是根本看不到占位图效果的。因为Glide有非常强大的缓存机制，我们刚才加载图片的时候Glide自动就已经将它缓存下来了，下次加载的时候将会直接从缓存中读取，不会再去找网络下载了，因而加载的速度非常快，所以占位图可能根本来不及显示。

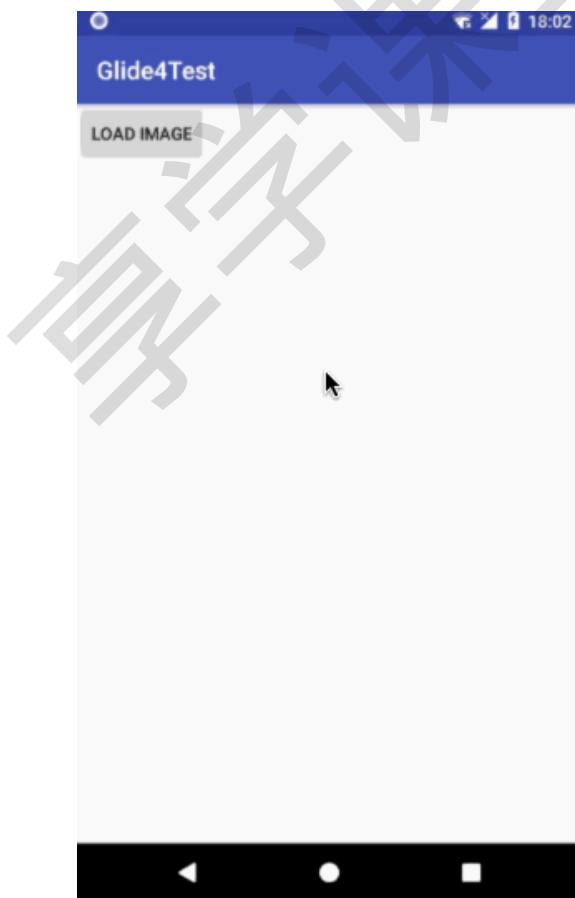
因此这里我们还需要稍微做一点修改，来让占位图能有机会显示出来，修改代码如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.loading)
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

可以看到，这里在RequestOptions对象中又串接了一个diskCacheStrategy()方法，并传入DiskCacheStrategy.NONE参数，这样就可以禁用掉Glide的缓存功能。

关于Glide缓存方面的内容我们待会儿会进行更详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：



可以看到，当点击Load Image按钮之后会立即显示一张占位图，然后等真正的图片加载完成之后会将占位图替换掉。

除了这种加载占位图之外，还有一种异常占位图。异常占位图就是指，如果因为某些异常情况导致图片加载失败，比如说手机网络信号不好，这个时候就显示这张异常占位图。

异常占位图的用法相信你已经可以猜到了，首先准备一张error.jpg图片，然后修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.ic_launcher_background)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

很简单，这里又串接了一个error()方法就可以指定异常占位图了。

其实看到这里，如果你熟悉Glide 3的话，相信你已经掌握Glide 4的变化规律了。在Glide 3当中，像placeholder()、error()、diskCacheStrategy()等等一系列的API，都是直接串联在Glide三步走方法中使用的。

而Glide 4中引入了一个RequestOptions对象，将这一系列的API都移动到了RequestOptions当中。这样做好处是可以使我们摆脱冗长的Glide加载语句，而且还能进行自己的API封装，因为RequestOptions是可以作为参数传入到方法中的。

比如你就可以写出这样的Glide加载工具类：

```
public class Glideutil {  
  
    public static void load(Context context,  
                           String url,  
                           ImageView imageView,  
                           RequestOptions options) {  
        Glide.with(context)  
            .load(url)  
            .apply(options)  
            .into(imageView);  
    }  
  
}
```

指定图片大小

实际上，使用Glide在大多数情况下我们都是不需要指定图片大小的，因为Glide会自动根据ImageView的大小来决定图片的大小，以此保证图片不会占用过多的内存从而引发OOM。

不过，如果你真的有这样的需求，必须给图片指定一个固定的大小，Glide仍然是支持这个功能的。修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()  
    .override(200, 100);  
Glide.with(this)  
    .load(url)  
    .apply(options)  
    .into(imageView);
```

仍然非常简单，这里使用override()方法指定了一个图片的尺寸。也就是说，Glide现在只会将图片加载成200*100像素的尺寸，而不管你的ImageView的大小是多少了。

如果你想加载一张图片的原始尺寸的话，可以使用Target.SIZE_ORIGINAL关键字，如下所示：

```
RequestOptions options = new RequestOptions()
    .override(Target.SIZE_ORIGINAL);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

这样的话，Glide就不会再去自动压缩图片，而是会去加载图片的原始尺寸。当然，这种写法也会面临着更高的OOM风险。

缓存机制

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就来分别学习一下这两种缓存的使用方法。

首先来看内存缓存。

你要知道，默认情况下，Glide自动就是开启内存缓存的。也就是说，当我们使用Glide加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用Glide再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个RecyclerView当中反复上下滑动，RecyclerView中只要是Glide加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而Glide最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为Glide默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide对此提供了接口：

```
RequestOptions options = new RequestOptions()
    .skipMemoryCache(true);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

可以看到，只需要调用skipMemoryCache()方法并传入true，就表示禁用掉Glide的内存缓存功能。

接下来我们开始学习硬盘缓存方面的内容。

其实在刚刚学习占位图功能的时候，我们就使用过硬盘缓存的功能了。当时为了禁止Glide对图片进行硬盘缓存而使用了如下代码：

```
RequestOptions options = new RequestOptions()
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

调用diskCacheStrategy()方法并传入DiskCacheStrategy.NONE，就可以禁用掉Glide的硬盘缓存功能了。

这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收五种参数：

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.DATA： 表示只缓存原始图片。
- DiskCacheStrategy.RESOURCE： 表示只缓存转换过后的图片。
- DiskCacheStrategy.ALL： 表示既缓存原始图片，也缓存转换过后的图片。
- DiskCacheStrategy.AUTOMATIC： 表示让Glide根据图片资源智能地选择使用哪一种缓存策略（默认选项）。

其中，DiskCacheStrategy.DATA对应Glide 3中的DiskCacheStrategy.SOURCE，DiskCacheStrategy.RESOURCE对应Glide 3中的DiskCacheStrategy.RESULT。而DiskCacheStrategy.AUTOMATIC是Glide 4中新增的一种缓存策略，并且在不指定diskCacheStrategy的情况下默认使用就是的这种缓存策略。

上面五种参数的解释本身并没有什么难理解的地方，但是关于转换过后的图片这个概念大家可能需要了解一下。就是当我们使用Glide去加载一张图片的时候，Glide默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在稍后学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。

指定加载格式

我们都知道，Glide其中一个非常亮眼的功能就是可以加载GIF图片，而同样作为非常出色的图片加载框架的Picasso是不支持这个功能的。

而且使用Glide加载GIF图并不需要编写什么额外的代码，Glide内部会自动判断图片格式。比如我们将加载图片的URL地址改成一张GIF图，如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/test.gif")
    .into(imageview);
```

现在重新运行一下代码，效果如下图所示：



也就是说，不管我们传入的是一张普通图片，还是一张GIF图片，Glide都会自动进行判断，并且可以正确地把它解析并展示出来。

但是如果我想指定加载格式该怎么办呢？就比如说，我希望加载的这张图必须是一张静态图片，我不需要Glide自动帮我判断它到底是静图还是GIF图。

想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
Glide.with(this)
    .asBitmap()
    .load("http://guolin.tech/test.gif")
    .into(imageview);
```

可以看到，这里在with()方法的后面加入了一个asBitmap()方法，这个方法的意思就是说这里只允许加载静态图片，不需要Glide去帮我们自动进行图片格式的判断了。如果你传入的还是一张GIF图的话，Glide会展示这张GIF图的第一帧，而不会去播放它。

熟悉Glide 3的朋友对asBitmap()方法肯定不会陌生对吧？但是千万不要觉得这里就没有陷阱了，在Glide 3中的语法是先load()再asBitmap()的，而在Glide 4中是先asBitmap()再load()的。乍一看可能分辨不出来有什么区别，但如果你写错了顺序就肯定会报错了。

那么类似地，既然我们能强制指定加载静态图片，就也能强制指定加载动态图片，对应的方法是asGif()。而Glide 4中又新增了asFile()方法和asDrawable()方法，分别用于强制指定文件格式的加载和Drawable格式的加载，用法都比较简单，就不再进行演示了。

回调与监听

回调与监听这部分的内容稍微有点多，我们分成四部分来学习一下。

1. into()方法

我们都知道Glide的into()方法中是可以传入ImageView的。那么into()方法还可以传入别的参数吗？我们可以让Glide加载出来的图片不显示到ImageView上吗？答案是肯定的，这就需要用到自定义Target功能。

Glide中的Target功能多样且复杂，下面我就先简单演示一种SimpleTarget的用法吧，代码如下所示：

```
SimpleTarget<Drawable> simpleTarget = new
SimpleTarget<Drawable>() {
    @Override
    public void onResourceReady(Drawable resource,
Transition<? super Drawable> transition) {
        imageView.setImageDrawable(resource);
    }
};

public void loadImage(View view) {
    Glide.with(this)
        .load("http://guolin.tech/book.png")
        .into(simpleTarget);
}
```

这里我们创建了一个SimpleTarget的实例，并且指定它的泛型是Drawable，然后重写了onResourceReady()方法。在onResourceReady()方法中，我们就可以获取到Glide加载出来的图片对象了，也就是方法参数中传过来的Drawable对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了ImageView上。

SimpleTarget的实现创建好了，那么只需要在加载图片的时候将它传入到into()方法中就可以了。

2. preload()方法

Glide加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果说有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待漫长的网络加载时间了，这该怎么办呢？

不用担心，Glide专门给我们提供了预加载的接口，也就是preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换into()方法即可，如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .preload();
```

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为Glide会直接从缓存当中去读取图片并显示出来，代码如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .into(imageview);
```

3. submit()方法

一直以来，我们使用Glide都是为了将图片显示到界面上。虽然我们知道Glide会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实Glide将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到submit()方法了。

submit()方法其实就是对应的Glide 3中的downloadOnly()方法，和preload()方法类似，submit()方法也是可以替换into()方法的，不过submit()方法的用法明显要比preload()方法复杂不少。这个方法只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。submit()方法有两个方法重载：

- submit()
- submit(int width, int height)

其中submit()方法是用于下载原始尺寸的图片，而submit(int width, int height)则可以指定下载图片的尺寸。

这里就以submit()方法来举例。当调用了submit()方法后会立即返回一个FutureTarget对象，然后Glide会在后台开始下载图片文件。接下来我们调用FutureTarget的get()方法就可以去获取下载好的图片文件了，如果此时图片还没有下载完，那么get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

```
public void downloadImage() {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                String url =  
                    "http://www.guolin.tech/book.png";  
                final Context context =  
                    getApplicationContext();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }).start();  
}
```

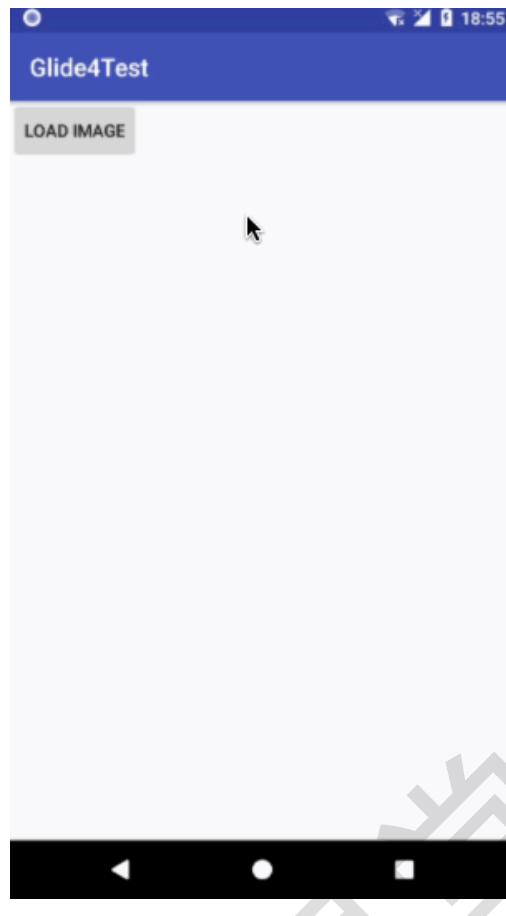
```
        FutureTarget<File> target =
Glide.with(context)
        .asFile()
        .load(url)
        .submit();
    final File imageFile = target.get();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(context,
imageFile.getPath(), Toast.LENGTH_LONG).show();
        }
    });
} catch (Exception e) {
    e.printStackTrace();
}
}).start();
}
```

这段代码稍微有一点点长，我带着大家解读一下。首先，submit()方法必须要用在子线程当中，因为刚才说了FutureTarget的get()方法是会阻塞线程的，因此这里的一步就是new了一个Thread。在子线程当中，我们先获取了一个Application Context，这个时候不能再用Activity作为Context了，因为会有Activity销毁了但子线程还没执行完这种可能出现。

接下来就是Glide的基本用法，只不过将into()方法替换成submit()方法，并且还使用了一个asFile()方法来指定加载格式。submit()方法会返回一个FutureTarget对象，这个时候其实Glide已经开始在后台下载图片了，我们随时都可以调用FutureTarget的get()方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的File对象返回。

最后，我们使用runOnUiThread()切回到主线程，然后使用Toast将下载好的图片文件路径显示出来。

现在重新运行一下代码，效果如下图所示。



这样我们就能清晰地看出来图片完整的缓存路径是什么了。

4. listener()方法

其实listener()方法的作用非常普遍，它可以用来监听Glide加载图片的状态。举个例子，比如说我们刚才使用了preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果Glide加载图片失败了，我该怎样调试错误的原因呢？答案都在listener()方法当中。

下面来看下listener()方法的基本用法吧，不同于刚才几个方法都是要替换into()方法的，listener()是结合into()方法一起使用的，当然也可以结合preload()方法一起使用。最基本的用法如下所示：

```
Glide.with(this)
    .load("http://www.guolin.tech/book.png")
    .listener(new RequestListener<Drawable>() {
        @Override
        public boolean onLoadFailed(@Nullable
GlideException e, Object model, Target<Drawable> target,
        boolean isFirstResource) {
            return false;
        }
    })
```

```
    @Override
    public boolean onResourceReady(Drawable
resource, Object model, Target<Drawable> target,
DataSource dataSource, boolean isFirstResource) {
        return false;
    }
})
.into(imageview);
```

这里我们在into()方法之前串接了一个listener()方法，然后实现了一个RequestListener的实例。其中RequestListener需要实现两个方法，一个onResourceReady()方法，一个onLoadFailed()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调onResourceReady()方法，而当图片加载失败的时候就会回调onLoadFailed()方法，onLoadFailed()方法中会将失败的GlideException参数传进来，这样我们就可以定位具体失败的原因了。

没错，listener()方法就是这么简单。不过还有一点需要处理，onResourceReady()方法和onLoadFailed()方法都有一个布尔值的返回值，返回false就表示这个事件没有被处理，还会继续向下传递，返回true就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单的例子，如果我们在RequestListener的onResourceReady()方法中返回了true，那么就不会再回调Target的onResourceReady()方法了。

图片变换

图片变换的意思就是说，Glide从加载了原始图片到最终展示给用户之前，又进行了一些变换处理，从而能够实现一些更加丰富的图片效果，如图片圆角化、圆形化、模糊化等等。

添加图片变换的用法非常简单，我们只需要在RequestOptions中串接transforms()方法，并将想要执行的图片变换操作作为参数传入transforms()方法即可，如下所示：

```
RequestOptions options = new RequestOptions()
    .transforms(...);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

至于具体要进行什么样的图片变换操作，这个通常都是需要我们自己来写的。不过Glide已经内置了几种图片变换操作，我们可以直接拿来使用，比如CenterCrop、FitCenter、CircleCrop等。

但所有的内置图片变换操作其实都不需要使用transform()方法，Glide为了方便我们使用直接提供了现成的API：

```
RequestOptions options = new RequestOptions()
    .centerCrop();

RequestOptions options = new RequestOptions()
    .fitCenter();

RequestOptions options = new RequestOptions()
    .circleCrop();
```

当然，这些内置的图片变换API其实也只是对transform()方法进行了一层封装而已，它们背后的源码仍然还是借助transform()方法来实现的。

这里我们就选择其中一种内置的图片变换操作来演示一下吧，circleCrop()方法是用来对图片进行圆形化裁剪的，我们动手试一下，代码如下所示：

```
String url = "http://guolin.tech/book.png";
RequestOptions options = new RequestOptions()
    .circleCrop();
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

重新运行一下程序并点击加载图片按钮，效果如下图所示。



可以看到，现在展示的图片是对原图进行圆形化裁剪后得到的图片。

当然，除了使用内置的图片变换操作之外，我们完全可以自定义自己的图片变换操作。理论上，在对图片进行变换这个步骤中我们可以进行任何的操作，你想对图片怎么样都可以。包括圆角化、圆形化、黑白化、模糊化等等，甚至你将原图片完全替换成另外一张图都是可以的。

不过由于这部分内容相对于Glide 3没有任何的变化，因此就不再重复进行讲解了。想学习自定义图片变换操作的朋友们可以参考这篇文章 [Android 图片加载框架最全解析（五），Glide强大的图片变换功能](#)。

关于图片变换，最后我们再来看一个非常优秀的开源库，`glide-transformations`。它实现了很多通用的图片变换效果，如裁剪变换、颜色变换、模糊变换等等，使得我们可以非常轻松地进行各种各样的图片变换。

`glide-transformations`的项目主页地址是 <https://github.com/wasabeef/glide-transformations>。

下面我们就来体验一下这个库的强大功能吧。首先需要将这个库引入到我们的项目当中，在app/build.gradle文件当中添加如下依赖：

```
dependencies {  
    implementation 'jp.wasabeef:glide-  
    transformations:3.0.1'  
}
```

我们可以对图片进行单个变换处理，也可以将多种图片变换叠加在一起使用。比如我想同时对图片进行模糊化和黑白化处理，就可以这么写：

```
String url = "http://guolin.tech/book.png";  
RequestOptions options = new RequestOptions()  
    .transforms(new BlurTransformation(), new  
    GrayscaleTransformation());  
Glide.with(this)  
    .load(url)  
    .apply(options)  
    .into(imageview);
```

可以看到，同时执行多种图片变换的时候，只需要将它们都传入到transforms()方法中即可。现在重新运行一下程序，效果如下图所示。



当然，这只是glide-transformations库的一小部分功能而已，更多的图片变换效果你可以到它的GitHub项目主页去学习。

自定义模块

自定义模块属于Glide中的高级功能，同时也是难度比较高的一部分内容。

这里我不可能在这一篇文章中将自定义模块的内容全讲一遍，限于篇幅的限制我只能讲一讲Glide 4中变化的这部分内容。关于Glide自定义模块的全部内容，请大家去参考 [Android图片加载框架最全解析（六），探究Glide的自定义模块功能](#) 这篇文章。

自定义模块功能可以将更改Glide配置，替换Glide组件等操作独立出来，使得我们能轻松地对Glide的各种配置进行自定义，并且又和Glide的图片加载逻辑没有任何交集，这也是一种低耦合编程方式的体现。下面我们就来学习一下自定义模块要如何实现。

首先定义一个我们自己的模块类，并让它继承自AppGlideModule，如下所示：

```
@GlideModule
```

```
public class MyAppGlideModule extends AppGlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {

    }

    @Override
    public void registerComponents(Context context, Glide
    glide, Registry registry) {

    }

}
```

可以看到，在MyAppGlideModule类当中，我们重写了applyOptions()和registerComponents()方法，这两个方法分别就是用来更改Glide配置以及替换Glide组件的。

注意在MyAppGlideModule类在上面，我们加入了一个@GlideModule的注解，这是Glide 4和Glide 3最大的一个不同之处。在Glide 3中，我们定义了自定义模块之后，还必须在AndroidManifest.xml文件中去注册它才能生效，而在Glide 4中是不需要的，因为@GlideModule这个注解已经能够让Glide识别到这个自定义模块了。

这样的话，我们就将Glide自定义模块的功能完成了。后面只需要在applyOptions()和registerComponents()这两个方法中加入具体的逻辑，就能实现更改Glide配置或者替换Glide组件的功能了。详情还是请参考Android图片加载框架最全解析（六），探究Glide的自定义模块功能这篇文章，这里就不再展开讨论了。

使用Generated API

Generated API是Glide 4中全新引入的一个功能，它的工作原理是使用注解处理器(Annotation Processor)来生成出一个API，在Application模块中可使用该流式API一次性调用到RequestBuilder，RequestOptions和集成库中所有的选项。

这么解释有点拗口，简单点说，就是Glide 4仍然给我们提供了一套和Glide 3一模一样的流式API接口。毕竟有些人还是觉得Glide 3的API更好用一些，比如说我。

Generated API对于熟悉Glide 3的朋友来说那是再简单不过了，基本上就是和Glide 3一模一样的用法，只不过需要把Glide关键字替换成GlideApp关键字，如下所示：

```
GlideApp.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .skipMemoryCache(true)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .override(Target.SIZE_ORIGINAL)
    .circleCrop()
    .into(imageView);
```

不过，有可能你的IDE中会提示找不到GlideApp这个类。这个类是通过编译时注解自动生成的，首先确保你的代码中有一个自定义的模块，并且给它加上了@GlideModule注解，也就是我们在上一节所讲的内容。然后在Android Studio中点击菜单栏Build -> Rebuild Project，GlideApp这个类就会自动生成了。

当然，Generated API所能做到的并不只是这些而已，它还可以对现有的API进行扩展，定制出任何属于你自己的API。

下面我来具体举个例子，比如说我们要求项目中所有图片的缓存策略全部都要缓存原始图片，那么每次在使用Glide加载图片的时候，都去指定diskCacheStrategy(DiskCacheStrategy.DATA)这么长长的一串代码，确实是让人比较心烦。这种情况我们就可以去定制一个自己的API了。

定制自己的API需要借助@GlideExtension和@GlideOption这两个注解。创建一个我们自定义的扩展类，代码如下所示：

```
@GlideExtension
public class MyGlideExtension {

    private MyGlideExtension() {

    }

    @GlideOption
    public static void cacheSource(RequestOptions
options) {

    options.diskCacheStrategy(DiskCacheStrategy.DATA);
}

}
```

这里我们定义了一个MyGlideExtension类，并且给加上了一个@GlideExtension注解，然后要将这个类的构造函数声明成private，这都是必须要求的写法。

接下来就可以开始自定义API了，这里我们定义了一个cacheSource()方法，表示只缓存原始图片，并给这个方法加上了@GlideOption注解。注意自定义API的方法都必须是静态方法，而且第一个参数必须是RequestOptions，后面你可以加入任意多个你想自定义的参数。

在cacheSource()方法中，我们仍然还是调用的diskCacheStrategy(DiskCacheStrategy.DATA)方法，所以说cacheSource()就是一层简化API的封装而已。

然后在Android Studio中点击菜单栏Build -> Rebuild Project，神奇的事情就会发生了，你会发现你已经可以使用这样的语句来加载图片了：

```
GlideApp.with(this)
    .load(url)
    .cacheSource()
    .into(imageView);
```

有了这个强大的功能之后，我们使用Glide就能变得更加灵活了。

结束语

这样我们基本上就将Glide 4的所有重要内容都介绍完了，如果你以前非常熟悉Glide 3的话，看完这篇文章之后相信你已经能够熟练使用Glide 4了。而如果你以前并未接触过Glide，仅仅只看这一篇文章可能了解得还不够深入，建议最好还是把前面的七篇文章也去通读一下，这样你才能成为一名Glide好手。

3.10 Android图片加载框架最全解析（一），Glide的基本用法

现在Android上的图片加载框架非常成熟，从最早的老牌图片加载框架UniversalImageLoader，到后来Google推出的Volley，再到后来的新星Glide和Picasso，当然还有Facebook的Fresco。每一个都非常稳定，功能也都十分强大。但是它们的使用场景基本都是重合的，也就是说我们基本只需要选择其中一个来进行学习和使用就足够了，每一个框架都尝试去掌握的话则有些浪费时间。

在这几个框架当中，我对Volley和Glide研究得比较深入，对UniversalImageLoader、Picasso和Fresco都只是有一些基本的了解。从易用性上来讲，Glide和Picasso应该都是完胜其他框架的，这两个框架都实在是太简单好用了，大多数情况下加载图片都是一行代码就能解决的，而UniversalImageLoader和Fresco则在这方面略逊一些。

那么再拿Glide和Picasso对比呢，首先这两个框架的用法非常相似，但其实它们各有特色。Picasso比Glide更加简洁和轻量，Glide比Picasso功能更为丰富。之前已经有人对这两个框架进行过全方面的对比，大家如果想了解更多的话可以去参考一下[这篇文章](#)。

总之，没有最好的框架，只有最适合自己的框架。经过多方面对比之后，我还是决定选择了Glide来进行研究，并且这也是Google官方推荐的图片加载框架。

说实话，关于Glide的文章我已经筹备了好久，去年这个时候本来打算要写了，但是一直都没有动笔。因为去年我的大部分时间都放在了写[《第二行代码》](#)上面，只能用碎片时间来写写博客，但是Glide的难度远超出了我用碎片时间所能掌握的难度。当然，这里我说的是对它的源码进行解

析的难度，不是使用上的难度，Glide的用法是很简单的。所以，我觉得去年我写不好Glide这个题材的文章，也就一直拖到了今年。

而现在，我花费了大量的精力去研究Glide的源码和各种用法，相信现在已经可以将它非常好地掌握了，因此我准备将我掌握的这些知识整理成一个新的系列，帮忙大家更好地学习Glide。这个Glide系列大概会有8篇左右文章，预计花半年时间写完，将会包括Glide的基本用法、源码解析、高级用法、功能扩展等内容，可能会是目前互联网上最详尽的Glide教程。

那么本篇文章是这个系列的第一篇文章，我们先来了解一下Glide的基本用法吧。

开始

Glide是一款由Bump Technologies开发的图片加载框架，使得我们可以在Android平台上以极度简单的方式加载和展示图片。

目前，Glide最新的稳定版本是3.7.0，虽然4.0已经推出RC版了，但是暂时问题还比较多。因此，我们这个系列的博客都会使用Glide 3.7.0版本来进行讲解，这个版本的Glide相当成熟和稳定。

要想使用Glide，首先需要将这个库引入到我们的项目当中。新建一个GlideTest项目，然后在app/build.gradle文件当中添加如下依赖：

```
dependencies {
    compile 'com.github.bumptech.glide:glide:3.7.0'
}
```

如果你还在使用Eclipse，可以点击[这里](#)下载Glide的jar包。

另外，Glide中需要用到网络功能，因此你还得在AndroidManifest.xml中声明一下网络权限才行：

```
<uses-permission
    android:name="android.permission.INTERNET" />
```

就是这么简单，然后我们就可以自由地使用Glide中的任意功能了。

加载图片

现在我们就来尝试一下如何使用Glide来加载图片吧。比如这是必应上一张首页美图的地址：

```
http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-CN10736487148_1920x1080.jpg
```

然后我们想要在程序当中去加载这张图片。

那么首先打开项目的布局文件，在布局当中加入一个Button和一个ImageView，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Image"
        android:onClick="loadImage"
        />

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

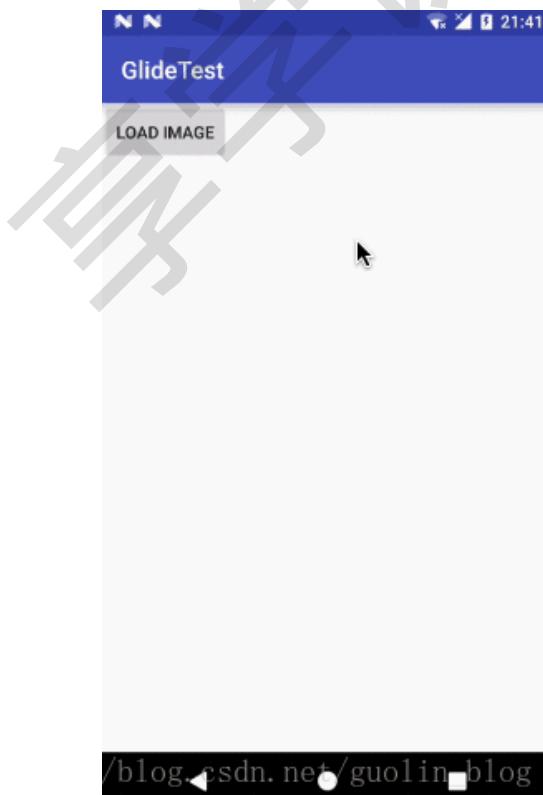
为了让用户点击Button的时候能够将刚才的图片显示在ImageView上，我们需要修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    ImageView imageView;
```

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    imageView = (ImageView)  
findViewById(R.id.image_view);  
}  
  
public void loadImage(View view) {  
    String url =  
"http://cn.bing.com/az/hprichbg/rb/Dongdaemun_ZH-  
CN10736487148_1920x1080.jpg";  
    Glide.with(this).load(url).into(imageView);  
}  
}
```

没错，就是这么简单。现在我们来运行一下程序，效果如下图所示：



可以看到，一张网络上的图片已经被成功下载，并且展示到ImageView上了。

而我们到底做了什么？实际上核心的代码就只有这一行而已：

```
Glide.with(this).load(url).into(imageview);
```

千万不要小看这一行代码，实际上仅仅就这一行代码，你已经可以做非常多非常多的事情了，包括加载网络上的图片、加载手机本地的图片、加载应用资源中的图片等等。

下面我们就来详细解析一下这行代码。

首先，调用Glide.with()方法用于创建一个加载图片的实例。with()方法可以接收Context、Activity或者Fragment类型的参数。也就是说我们选择的范围非常广，不管是在Activity还是Fragment中调用with()方法，都可以直接传this。那如果调用的地方既不在Activity中也不在Fragment中呢？也没关系，我们可以获取当前应用程序的ApplicationContext，传入到with()方法当中。注意with()方法中传入的实例会决定Glide加载图片的生命周期，如果传入的是Activity或者Fragment的实例，那么当这个Activity或Fragment被销毁的时候，图片加载也会停止。如果传入的是ApplicationContext，那么只有当应用程序被杀掉的时候，图片加载才会停止。

接下来看一下load()方法，这个方法用于指定待加载的图片资源。Glide支持加载各种各样的图片资源，包括网络图片、本地图片、应用资源、二进制流、Uri对象等等。因此load()方法也有很多个方法重载，除了我们刚才使用的加载一个字符串网址之外，你还可以这样使用load()方法：

```
// 加载本地图片  
File file = new File(getExternalCacheDir() +  
"/image.jpg");  
Glide.with(this).load(file).into(imageview);  
  
// 加载应用资源  
int resource = R.drawable.image;  
Glide.with(this).load(resource).into(imageview);  
  
// 加载二进制流  
byte[] image = getImageBytes();  
Glide.with(this).load(image).into(imageview);
```

```
// 加载Uri对象  
Uri imageUri = getImageUri();  
Glide.with(this).load(imageUri).into(imageview);
```

最后看一下into()方法，这个方法就很简单了，我们希望让图片显示在哪个ImageView上，把这个ImageView的实例传进去就可以了。当然，into()方法不仅仅是只能接收ImageView类型的参数，还支持很多更丰富的用法，不过那个属于高级技巧，我们会在后面的文章当中学习。

那么回顾一下Glide最基本的使用方式，其实就是关键的三步走：先with(), 再load(), 最后into()。熟记这三步，你就已经入门Glide了。

占位图

现在我们来学一些Glide的扩展内容。其实刚才所学的三步走就是Glide最核心的东西，而我们后面所要学习的所有东西都是在这个三步走的基础上不断进行扩展而已。

观察刚才加载网络图片的效果，你会发现，点击了Load Image按钮之后，要稍微等一会图片才会显示出来。这其实很容易理解，因为从网络上下载图片本来就是需要时间的。那么我们有没有办法再优化一下用户体验呢？当然可以，Glide提供了各种各样非常丰富的API支持，其中就包括了占位图功能。

顾名思义，占位图就是指在图片的加载过程中，我们先显示一张临时的图片，等图片加载出来了再替换成要加载的图片。

下面我们就来学习一下Glide占位图功能的使用方法，首先我事先准备好了一张loading.jpg图片，用来作为占位图显示。然后修改Glide加载部分的代码，如下所示：

```
Glide.with(this)  
    .load(url)  
    .placeholder(R.drawable.loading)  
    .into(imageview);
```

没错，就是这么简单。我们只是在刚才的三步走之间插入了一个placeholder()方法，然后将占位图片的资源id传入到这个方法中即可。另外，这个占位图的用法其实也演示了Glide当中绝大多数API的用法，其实就是在load()和into()方法之间串接任意想添加的功能就可以了。

不过如果你现在重新运行一下代码并点击Load Image，很可能是根本看不到占位图效果的。因为Glide有非常强大的缓存机制，我们刚才加载那张必应美图的时候Glide自动就已经将它缓存下来了，下次加载的时候将会直接从缓存中读取，不会再去网络下载了，因而加载的速度非常快，所以占位图可能根本来不及显示。

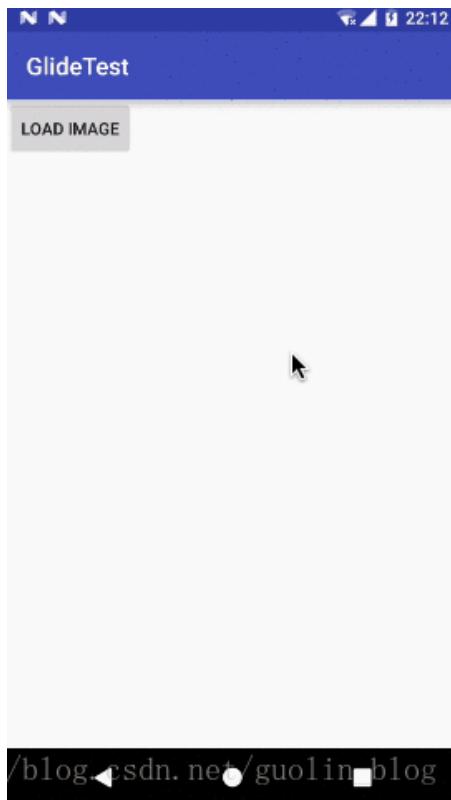
因此这里我们还需要稍微做一点修改，来让占位图能有机会显示出来，修改代码如下所示：

```
Glide.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

可以看到，这里串接了一个diskCacheStrategy()方法，并传入DiskCacheStrategy.NONE参数，这样就可以禁用掉Glide的缓存功能。

关于Glide缓存方面的内容我们将会在后面的文章进行详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：



可以看到，当点击Load Image按钮之后会立即显示一张占位图，然后等真正的图片加载完成之后会将占位图替换掉。

当然，这只是占位图的一种，除了这种加载占位图之外，还有一种异常占位图。异常占位图就是指，如果因为某些异常情况导致图片加载失败，比如说手机网络信号不好，这个时候就显示这张异常占位图。

异常占位图的用法相信你已经可以猜到了，首先准备一张error.jpg图片，然后修改Glide加载部分的代码，如下所示：

```
Glide.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

很简单，这里又串接了一个error()方法就可以指定异常占位图了。

现在你可以将图片的url地址修改成一个不存在的图片地址，或者干脆直接将手机的网络给关了，然后重新运行程序，效果如下图所示：



这样我们就把Glide提供的占位图功能都掌握了。

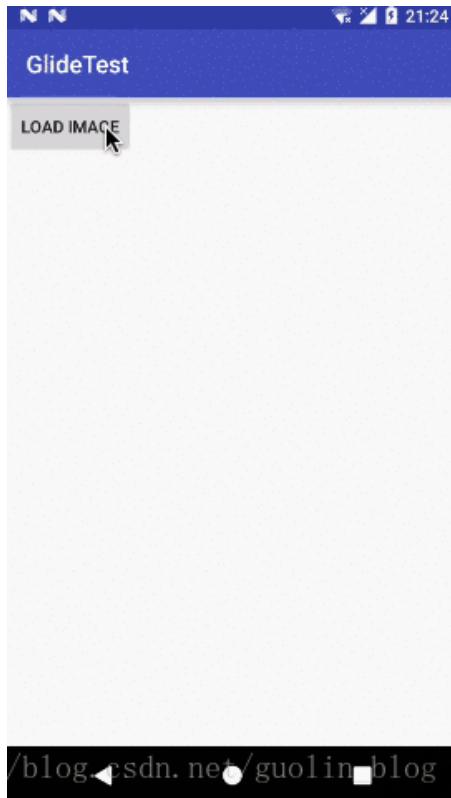
指定图片格式

我们还需要再了解一下Glide另外一个强大的功能，那就是Glide是支持加载GIF图片的。这一点确实非常牛逼，因为相比之下Jake Warton曾经明确表示过，Picasso是不会支持加载GIF图片的。

而使用Glide加载GIF图并不需要编写什么额外的代码，Glide内部会自动判断图片格式。比如这是一张GIF图片的URL地址：

```
http://p1.pstatp.com/large/166200019850062839d3
```

我们只需要将刚才那段加载图片代码中的URL地址替换成上面的地址就可以了，现在重新运行一下代码，效果如下图所示：



也就是说，不管我们传入的是一张普通图片，还是一张GIF图片，Glide都会自动进行判断，并且可以正确地把它解析并展示出来。

但是如果我想指定图片的格式该怎么办呢？就比如说，我希望加载的这张图必须是一张静态图片，我不需要Glide自动帮我判断它到底是静图还是GIF图。

想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
Glide.with(this)
    .load(url)
    .asBitmap()
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

可以看到，这里在load()方法的后面加入了一个asBitmap()方法，这个方法的意思就是说这里只允许加载静态图片，不需要Glide去帮我们自动进行图片格式的判断了。

现在重新运行一下程序，效果如下图所示：



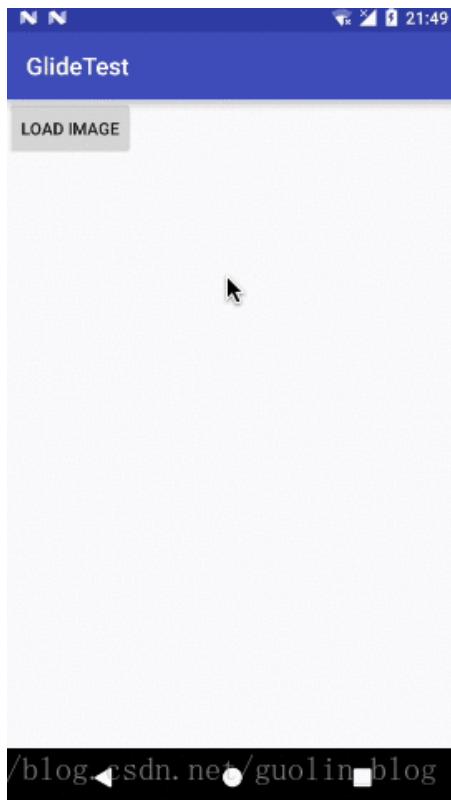
由于调用了asBitmap()方法，现在GIF图就无法正常播放了，而是会在界面上显示第一帧的图片。

那么类似地，既然我们能强制指定加载静态图片，就也能强制指定加载动态图片。比如说我们想要实现必须加载动态图片的功能，就可以这样写：

```
Glide.with(this)
    .load(url)
    .asGif()
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

这里调用了asGif()方法替代了asBitmap()方法，很好理解，相信不用我多做什么解释了。

那么既然指定了只允许加载动态图片，如果我们传入了一张静态图片的URL地址又会怎么样呢？试一下就知道了，将图片的URL地址改成刚才的必应美图，然后重新运行代码，效果如下图所示。



没错，如果指定了只能加载动态图片，而传入的图片却是一张静图的话，那么结果自然就只有加载失败喽。

指定图片大小

实际上，使用Glide在绝大多数情况下我们都是不需要指定图片大小的。

在学习本节内容之前，你可能还需要先了解一个概念，就是我们平时在加载图片的时候很容易会造成内存浪费。什么叫内存浪费呢？比如说一张图片的尺寸是1000*1000像素，但是我们界面上的ImageView可能只有200*200像素，这个时候如果你不对图片进行任何压缩就直接读取到内存中，这就属于内存浪费了，因为程序中根本就用不到这么高像素的图片。

关于图片压缩这方面，我之前也翻译过Android官方的一篇文章，感兴趣的朋友可以去阅读一下 [Android高效加载大图、多图解决方案，有效避免程序OOM](#)。

而使用Glide，我们就完全不用担心图片内存浪费，甚至是内存溢出的问题。因为Glide从来都不会直接将图片的完整尺寸全部加载到内存中，而是用多少加载多少。Glide会自动判断ImageView的大小，然后只将这么大的图片像素加载到内存当中，帮助我们节省内存开支。

当然，Glide也并没有使用什么神奇的魔法，它内部的实现原理其实就是上面那篇文章当中介绍的技术，因此掌握了最基本的实现原理，你也可以自己实现一套这样的图片压缩机制。

也正是因为Glide是如此的智能，所以刚才在开始的时候我就说了，在绝大多数情况下我们都是不需要指定图片大小的，因为Glide会自动根据ImageView的大小来决定图片的大小。

不过，如果你真的有这样的需求，必须给图片指定一个固定的大小，Glide仍然是支持这个功能的。修改Glide加载部分的代码，如下所示：

```
Glide.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .override(100, 100)
    .into(imageview);
```

仍然非常简单，这里使用override()方法指定了一个图片的尺寸，也就是说，Glide现在只会将图片加载成100*100像素的尺寸，而不会管你的ImageView的大小是多少了。

好了，今天是我们这个Glide系列的第一篇文章，写了这么多内容已经算是挺不错的了。现在你已经了解了Glide的基本用法，当然也有一些最常用的用法。下一篇文章当中，我们会尝试去分析Glide的源码，研究一下在这些基本用法的背后，Glide到底执行了什么神奇的操作，能够使得我们加载图片变得这么简单？感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（二），从源码的角度理解Glide的执行流程](#)。

3.11 Android图片加载框架最全解析（二），从源码的角度理解Glide的执行流程

在本系列的上一篇文章中，我们学习了Glide的基本用法，体验了这个图片加载框架的强大功能，以及它非常简便的API。还没有看过上一篇文章的朋友，建议先去阅读 [Android图片加载框架最全解析（一），Glide的基本用法](#)。

在多数情况下，我们想要在界面上加载并展示一张图片只需要一行代码就能实现，如下所示：

```
Glide.with(this).load(url).into(imageview);
```

虽说只有这简简单单的一行代码，但大家可能不知道的是，Glide在背后帮我们默默执行了成吨的工作。这个形容词我想了很久，因为我觉得用非常多这个形容词不足以描述Glide背后的工作量，我查到的英文资料是用 tons of work 来进行形容的，因此我觉得这里使用成吨来形容更加贴切一些。

虽说我们在平时使用Glide的时候格外地简单和方便，但是知其然也要知其所以然。那么今天我们就来解析一下Glide的源码，看看它在这些简单用法的背后，到底执行了多么复杂的工作。

如何阅读源码

在开始解析Glide源码之前，我想先和大家谈一下该如何阅读源码，这个问题也是我平时被问得比较多的，因为很多人都觉得阅读源码是一件比较困难的事情。

那么阅读源码到底困难吗？这个当然主要还是要视具体的源码而定。比如同样是图片加载框架，我读Volley的源码时就感觉酣畅淋漓，并且对Volley的架构设计和代码质量深感佩服。读Glide的源码时却让我相当痛苦，代码极其难懂。当然这里我并不是说Glide的代码写得不好，只是因为Glide和复杂程度和Volley完全不是在一个量级上的。

那么，虽然源码的复杂程度是外在的不可变条件，但我们却可以通过一些技巧来提升自己阅读源码的能力。这里我和大家分享一下我平时阅读源码时所使用的技巧，简单概括就是八个字：抽丝剥茧、点到即止。应该认准一个功能点，然后去分析这个功能点是如何实现的。但只要去追寻主体的实现逻辑即可，千万不要试图去搞懂每一行代码都是什么意思，那样很容易会陷入到思维黑洞当中，而且越陷越深。因为这些庞大的系统都不是由

一个人写出来的，每一行代码都想搞明白，就会感觉自己是在盲人摸象，永远也研究不透。如果只是去分析主体的实现逻辑，那么就有比较明确的目的性，这样阅读源码会更加轻松，也更加有成效。

而今天带大家阅读的Glide源码就非常适合使用这个技巧，因为Glide的源码太复杂了，千万不要试图去搞明白它每行代码的作用，而是应该只分析它的主体实现逻辑。那么我们本篇文章就先确立好一个目标，就是要通过阅读源码搞明白下面这行代码：

```
Glide.with(this).load(url).into(imageview);
```

到底是如何实现将一张网络图片展示到ImageView上面的。先将Glide的一整套图片加载机制的基本流程梳理清楚，然后我们再通过后面的几篇文章具体去了解Glide源码方方面面的细节。

准备好了吗？那么我们现在开始。

源码下载

既然是要阅读Glide的源码，那么我们自然需要先将Glide的源码下载下来。其实如果你是使用在build.gradle中添加依赖的方式将Glide引入到项目中的，那么源码自动就已经下载下来了，在Android Studio中就可以直接进行查看。

不过，使用添加依赖的方式引入的Glide，我们只能看到它的源码，但不能做任何的修改，如果你还需要修改它的源码的话，可以到GitHub上面将它的完整源码下载下来。

Glide的GitHub主页的地址是：<https://github.com/bumptech/glide>

不过在这个地址下载到的永远都是最新的源码，有可能还正在处于开发当中。而我们整个系列都是使用Glide 3.7.0这个版本来进行讲解的，因此如果你需要专门去下载3.7.0版本的源码，可以到这个地址进行下载：

<https://github.com/bumptech/glide/tree/v3.7.0>

开始阅读

我们在上一篇文章中已经学习过了，Glide最基本的用法就是三步走：先with()，再load()，最后into()。那么我们开始一步步阅读这三步走的源码，先从with()看起。

1. with()

with()方法是Glide类中的一组静态方法，它有好几个方法重载，我们来看一下Glide类中所有with()方法的方法重载：

```
public class Glide {  
    ...  
  
    public static RequestManager with(Context context) {  
        RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
        return retriever.get(context);  
    }  
  
    public static RequestManager with(Activity activity)  
{  
        RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
        return retriever.get(activity);  
    }  
  
    public static RequestManager with(FragmentActivity  
activity) {  
        RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
        return retriever.get(activity);  
    }  
  
    @TargetApi(Build.VERSION_CODES.HONEYCOMB)  
    public static RequestManager  
with(android.app.Fragment fragment) {  
        RequestManagerRetriever retriever =  
RequestManagerRetriever.get();  
        return retriever.get(fragment);  
    }  
}
```

```
public static RequestManager with(Fragment fragment)
{
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(fragment);
}
}
```

可以看到，with()方法的重载种类非常多，既可以传入Activity，也可以传入Fragment或者是Context。每一个with()方法重载的代码都非常简单，都是先调用RequestManagerRetriever的静态get()方法得到一个RequestManagerRetriever对象，这个静态get()方法就是一个单例实现，没什么需要解释的。然后再调用RequestManagerRetriever的实例get()方法，去获取RequestManager对象。

而RequestManagerRetriever的实例get()方法中的逻辑是什么样的呢？我们一起来看一看：

```
public class RequestManagerRetriever implements
Handler.Callback {

    private static final RequestManagerRetriever INSTANCE
= new RequestManagerRetriever();

    private volatile RequestManager applicationManager;

    ...

    /**
     * Retrieves and returns the RequestManagerRetriever
     singleton.
     */
    public static RequestManagerRetriever get() {
        return INSTANCE;
    }

    private RequestManager getApplicationManager(Context
context) {
```

```
// Either an application context or we're on a
background thread.
    if (applicationManager == null) {
        synchronized (this) {
            if (applicationManager == null) {
                // Normally pause/resume is taken
                care of by the fragment we add to the fragment or
                activity.

                // However, in this case since the
                manager attached to the application will not receive
                lifecycle

                // events, we must force the manager
                to start resumed using ApplicationLifecycle.

                applicationManager = new
                RequestManager(context.getApplicationContext(),
                    new ApplicationLifecycle(),
                    new EmptyRequestManagerTreeNode());
            }
        }
    }
    return applicationManager;
}

public RequestManager get(Context context) {
    if (context == null) {
        throw new IllegalArgumentException("You
cannot start a load on a null Context");
    } else if (Util.isOnMainThread() && !(context
instanceof Application)) {
        if (context instanceof FragmentActivity) {
            return get((FragmentActivity) context);
        } else if (context instanceof Activity) {
            return get((Activity) context);
        } else if (context instanceof ContextWrapper)
{
            return get(((ContextWrapper)
context).getBaseContext());
        }
    }
}
```

```
        }

        return getApplicationManager(context);
    }

    public RequestManager get(FragmentActivity activity)
    {
        if (util.isOnBackgroundThread()) {
            return get(activity.getApplicationContext());
        } else {
            assertNotDestroyed(activity);
            FragmentManager fm =
activity.getSupportFragmentManager();
            return supportFragmentGet(activity, fm);
        }
    }

    public RequestManager get(Fragment fragment) {
        if (fragment.getActivity() == null) {
            throw new IllegalArgumentException("You
cannot start a load on a fragment before it is
attached");
        }
        if (util.isOnBackgroundThread()) {
            return
get(fragment.getActivity().getApplicationContext());
        } else {
            FragmentManager fm =
fragment.getChildFragmentManager();
            return
supportFragmentGet(fragment.getActivity(), fm);
        }
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    public RequestManager get(Activity activity) {
        if (util.isOnBackgroundThread() ||
Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
            return get(activity.getApplicationContext());
        }
```

```
    } else {
        assertNotDestroyed(activity);
        android.app.FragmentManager fm =
activity.getFragmentManager();
        return fragmentGet(activity, fm);
    }
}

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private static void assertNotDestroyed(Activity
activity) {
    if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.JELLY_BEAN_MR1 &&
activity.isDestroyed()) {
        throw new IllegalArgumentException("You
cannot start a load for a destroyed activity");
    }
}

@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
public RequestManager get(android.app.Fragment
fragment) {
    if (fragment.getActivity() == null) {
        throw new IllegalArgumentException("You
cannot start a load on a fragment before it is
attached");
    }
    if (util.isOnBackgroundThread() ||
Build.VERSION.SDK_INT <
Build.VERSION_CODES.JELLY_BEAN_MR1) {
        return
get(fragment.getActivity().getApplicationContext());
    } else {
        android.app.FragmentManager fm =
fragment.getChildFragmentManager();
        return fragmentGet(fragment.getActivity(),
fm);
    }
}
```

```
    }

    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
    RequestManagerFragment
    getRequestManagerFragment(final
        android.app.FragmentManager fm) {
        RequestManagerFragment current =
        (RequestManagerFragment)
        fm.findFragmentByTag(FRAGMENT_TAG);
        if (current == null) {
            current =
            pendingRequestManagerFragments.get(fm);
            if (current == null) {
                current = new RequestManagerFragment();
                pendingRequestManagerFragments.put(fm,
                current);
                fm.beginTransaction().add(current,
                FRAGMENT_TAG).commitAllowingStateLoss();
            }
        }
        handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER,
        fm).sendToTarget();
    }
    return current;
}

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
RequestManager fragmentGet(Context context,
    android.app.FragmentManager fm) {
    RequestManagerFragment current =
    getRequestManagerFragment(fm);
    RequestManager requestManager =
    current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context,
        current.getLifecycle(),
        current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
}
```

```
        }

        return requestManager;
    }

    SupportRequestManagerFragment
getSupportRequestManagerFragment(final FragmentManager
fm) {
    SupportRequestManagerFragment current =
(SupportRequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);
    if (current == null) {
        current =
pendingSupportRequestManagerFragments.get(fm);
        if (current == null) {
            current = new
SupportRequestManagerFragment();
            pendingSupportRequestManagerFragments.put(fm, current);
            fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();
        }
    }
    return current;
}

RequestManager supportFragmentGet(Context context,
FragmentManager fm) {
    SupportRequestManagerFragment current =
getSupportRequestManagerFragment(fm);
    RequestManager requestManager =
current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context,
current.getLifecycle(),
current.getRequestManagerTreeNode());
    }
}
```

```
        current.setRequestManager(requestManager);
    }
    return requestManager;
}

...
}
```

上述代码虽然看上去逻辑有点复杂，但是将它们梳理清楚后还是很简单的。RequestManagerRetriever类中看似有很多个get()方法的重载，什么Context参数，Activity参数，Fragment参数等等，实际上只有两种情况而已，即传入Application类型的参数，和传入非Application类型的参数。

我们先来看传入Application参数的情况。如果在Glide.with()方法中传入的是一个Application对象，那么这里就会调用带有Context参数的get()方法重载，然后会在第44行调用getApplicationManager()方法来获取一个RequestManager对象。其实这是最简单的一种情况，因为Application对象的生命周期即应用程序的生命周期，因此Glide并不需要做什么特殊的处理，它自动就是和应用程序的生命周期是同步的，如果应用程序关闭的话，Glide的加载也会同时终止。

接下来我们看传入非Application参数的情况。不管你在Glide.with()方法中传入的是Activity、FragmentActivity、v4包下的Fragment、还是app包下的Fragment，最终的流程都是一样的，那就是会向当前的Activity当中添加一个隐藏的Fragment。具体添加的逻辑是在上述代码的第117行和第141行，分别对应的app包和v4包下的两种Fragment的情况。那么这里为什么要添加一个隐藏的Fragment呢？因为Glide需要知道加载的生命周期。很简单的一个道理，如果你在某个Activity上正在加载着一张图片，结果图片还没加载出来，Activity就被用户关掉了，那么图片还应该继续加载吗？当然不应该。可是Glide并没有办法知道Activity的生命周期，于是Glide就使用了添加隐藏Fragment的这种小技巧，因为Fragment的生命周期和Activity是同步的，如果Activity被销毁了，Fragment是可以监听到的，这样Glide就可以捕获这个事件并停止图片加载了。

这里额外再提一句，从第48行代码可以看出，如果我们是在非主线程当中使用的Glide，那么不管你是传入的Activity还是Fragment，都会被强制当成Application来处理。不过其实这就属于是在分析代码的细节了，本篇文章我们将会把目光主要放在Glide的主线工作流程上面，后面不会过多去分析这些细节方面的内容。

总体来说，第一个with()方法的源码还是比较容易理解的。其实就是为了得到一个RequestManager对象而已，然后Glide会根据我们传入with()方法的参数来确定图片加载的生命周期，并没有什么特别复杂的逻辑。不过复杂的逻辑还在后面等着我们呢，接下来我们开始分析第二步，load()方法。

2. load()

由于with()方法返回的是一个RequestManager对象，那么很容易就能想到，load()方法是在RequestManager类当中的，所以说我们首先要看的就是RequestManager这个类。不过在上一篇文章中我们学过，Glide是支持图片URL字符串、图片本地路径等等加载形式的，因此RequestManager中也有很多个load()方法的重载。但是这里我们不可能把每个load()方法的重载都看一遍，因此我们就只选其中一个加载图片URL字符串的load()方法来进行研究吧。

RequestManager类的简化代码如下所示：

```
public class RequestManager implements LifecycleListener
{
    ...
    /**
     * Returns a request builder to load the given {@link String}.
     * signature.
     *
     * @see #fromString()
     * @see #load(Object)
     */
}
```

```
* @param string A file path, or a uri or url handled
by {@link com.bumptech.glide.load.model.UriLoader}.
*/
public DrawableTypeRequest<String> load(String
string) {
    return (DrawableTypeRequest<String>)
fromString().load(string);
}

/**
 * Returns a request builder that loads data from
{@link String}s using an empty signature.
*
* <p>
* Note - this method caches data using only the
given String as the cache key. If the data is a Uri
outside of
*      your control, or you otherwise expect the data
represented by the given String to change without the
String
*      identifier changing, Consider using
*      {@link GenericRequestBuilder#signature(Key)}
to mixin a signature
*      you create that identifies the data currently
at the given String that will invalidate the cache if
that data
*      changes. Alternatively, using {@link
DiskCacheStrategy#NONE} and/or
*      {@link
DrawableRequestBuilder#skipMemoryCache(boolean)} may be
appropriate.
* </p>
*
* @see #from(Class)
* @see #load(String)
*/
public DrawableTypeRequest<String> fromString() {
    return loadGeneric(String.class);
```

```
    }

    private <T> DrawableTypeRequest<T>
loadGeneric(Class<T> modelClass) {
    ModelLoader<T, InputStream> streamModelLoader =
Glide.buildStreamModelLoader(modelClass, context);
    ModelLoader<T, ParcelFileDescriptor>
fileDescriptorModelLoader =
Glide.buildFileDescriptorModelLoader(modelClass,
context);
    if (modelClass != null && streamModelLoader ==
null && fileDescriptorModelLoader == null) {
        throw new IllegalArgumentException("Unknown
type " + modelClass + ". You must provide a Model of a
type for"
        + " which there is a registered
ModelLoader, if you are using a custom model, you must
first call"
        + " Glide#register with a
ModelLoaderFactory for your custom model class");
    }
    return optionsApplier.apply(
        new DrawableTypeRequest<T>(modelClass,
streamModelLoader, fileDescriptorModelLoader, context,
        glide, requestTracker, lifecycle,
optionsApplier));
}

...
}
```

RequestManager类的代码是非常多的，但是经过我这样简化之后，看上去就比较清爽了。在我们只探究加载图片URL字符串这一个load()方法的情况下，那么比较重要的方法就只剩下上述代码中的这三个方法。

那么我们先来看load()方法，这个方法中的逻辑是非常简单的，只有一行代码，就是先调用了fromString()方法，再调用load()方法，然后把传入的图片URL地址传进去。而fromString()方法也极为简单，就是调用了loadGeneric()方法，并且指定参数为String.class，因为load()方法传入的是一个字符串参数。那么看上去，好像主要的工作都是在loadGeneric()方法中进行的了。

其实loadGeneric()方法也没几行代码，这里分别调用了Glide.buildStreamModelLoader()方法和Glide.buildFileDescriptorModelLoader()方法来获得ModelLoader对象。ModelLoader对象是用于加载图片的，而我们给load()方法传入不同类型的参数，这里也会得到不同的ModelLoader对象。不过buildStreamModelLoader()方法内部的逻辑还是蛮复杂的，这里就不展开介绍了，要不然篇幅实在收不住，感兴趣的话你可以自己研究。由于我们刚才传入的参数是String.class，因此最终得到的是StreamStringLoader对象，它是实现了ModelLoader接口的。

最后我们可以看到，loadGeneric()方法是要返回一个DrawableTypeRequest对象的，因此在loadGeneric()方法的最后又去new了一个DrawableTypeRequest对象，然后把刚才获得的ModelLoader对象，还有一大堆杂七杂八的东西都传了进去。具体每个参数的含义和作用就不解释了，我们只看主线流程。

那么这个DrawableTypeRequest的作用是什么呢？我们来看下它的源码，如下所示：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType> implements  
DownloadOptions {  
    private final ModelLoader<ModelType, InputStream>  
    streamModelLoader;  
    private final ModelLoader<ModelType,  
    ParcelFileDescriptor> fileDescriptorModelLoader;  
    private final RequestManager.OptionsApplier  
    optionsApplier;
```

```
    private static <A, Z, R> FixedLoadProvider<A,
ImageVideoWrapper, Z, R> buildProvider(Glide glide,
    ModelLoader<A, InputStream>
streamModelLoader,
    ModelLoader<A, ParcelFileDescriptor>
fileDescriptorModelLoader, Class<Z> resourceClass,
    Class<R> transcodedClass,
    ResourceTranscoder<Z, R> transcoder) {
    if (streamModelLoader == null &&
fileDescriptorModelLoader == null) {
        return null;
    }

    if (transcoder == null) {
        transcoder =
glide.buildTranscoder(resourceClass, transcodedClass);
    }
    DataLoadProvider<ImageVideoWrapper, Z>
dataLoadProvider =
glide.buildDataProvider(ImageVideoWrapper.class,
    resourceClass);
    ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
    fileDescriptorModelLoader);
    return new FixedLoadProvider<A,
ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
}

DrawableTypeRequest<Class<ModelType> modelClass,
ModelLoader<ModelType, InputStream> streamModelLoader,
    ModelLoader<ModelType, ParcelFileDescriptor>
fileDescriptorModelLoader, Context context, Glide glide,
    RequestTracker requestTracker, Lifecycle
lifecycle, RequestManager.OptionsApplier optionsApplier)
{
    super(context, modelClass,
```

```
        buildProvider(glide, streamModelLoader,
fileDescriptorModelLoader, GifBitmapWrapper.class,
                GlideDrawable.class, null),
                glide, requestTracker, lifecycle);
this.streamModelLoader = streamModelLoader;
this.fileDescriptorModelLoader =
fileDescriptorModelLoader;
this.optionsApplier = optionsApplier;
}

/**
 * Attempts to always load the resource as a {@link
android.graphics.Bitmap}, even if it could actually be
animated.
*
* @return A new request builder for loading a {@link
android.graphics.Bitmap}
*/
public BitmapTypeRequest<ModelType> asBitmap() {
    return optionsApplier.apply(new
BitmapTypeRequest<ModelType>(this, streamModelLoader,
                fileDescriptorModelLoader,
optionsApplier));
}

/**
 * Attempts to always load the resource as a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
* <p>
* If the underlying data is not a GIF, this will
fail. As a result, this should only be used if the model
* represents an animated GIF and the caller
wants to interact with the GifDrawable directly. Normally
using
* just an {@link DrawableTypeRequest} is
sufficient because it will determine whether or
* not the given data represents an animated GIF
and return the appropriate animated or not animated
```

```
*      {@link android.graphics.drawable.Drawable}
automatically.

* </p>
*
* @return A new request builder for loading a {@link
com.bumptech.glide.load.resource.gif.GifDrawable}.
*/
public GifTypeRequest<ModelType> asGif() {
    return optionsApplier.apply(new
GifTypeRequest<ModelType>(this, streamModelLoader,
optionsApplier));
}

...
}
```

这个类中的代码本身就不多，我只是稍微做了一点简化。可以看到，最主要的就是它提供了asBitmap()和asGif()这两个方法。这两个方法我们在上一篇文章当中都是学过的，分别是用于强制指定加载静态图片和动态图片。而从源码中可以看出，它们分别又创建了一个BitmapTypeRequest和GifTypeRequest，如果没有进行强制指定的话，那默认就是使用DrawableTypeRequest。

好的，那么我们再回到RequestManager的load()方法中。刚才已经分析过了，fromString()方法会返回一个DrawableTypeRequest对象，接下来会调用这个对象的load()方法，把图片的URL地址传进去。但是我们刚才看到了，DrawableTypeRequest中并没有load()方法，那么很容易就能猜想到，load()方法是在父类当中的。

DrawableTypeRequest的父类是DrawableRequestBuilder，我们来看下这个类的源码：

```
public class DrawableRequestBuilder<ModelType>
    extends GenericRequestBuilder<ModelType,
ImageVideoWrapper, GifBitmapWrapper, GlideDrawable>
    implements BitmapOptions, DrawableOptions {
```

```
    DrawableRequestBuilder(Context context,
    Class<ModelType> modelClass,
        LoadProvider<ModelType, ImageVideoWrapper,
    GifBitmapWrapper, GlideDrawable> loadProvider, Glide
    glide,
        RequestTracker requestTracker, Lifecycle
    lifecycle) {
    super(context, modelClass, loadProvider,
    GlideDrawable.class, glide, requestTracker, lifecycle);
    // Default to animating.
    crossFade();
}

public DrawableRequestBuilder<ModelType> thumbnail(
    DrawableRequestBuilder<?> thumbnailRequest) {
super.thumbnail(thumbnailRequest);
return this;
}

@Override
public DrawableRequestBuilder<ModelType> thumbnail(
    GenericRequestBuilder<?, ?, ?, GlideDrawable>
thumbnailRequest) {
super.thumbnail(thumbnailRequest);
return this;
}

@Override
public DrawableRequestBuilder<ModelType>
thumbnail(float sizeMultiplier) {
super.thumbnail(sizeMultiplier);
return this;
}

@Override
public DrawableRequestBuilder<ModelType>
sizeMultiplier(float sizeMultiplier) {
super.sizeMultiplier(sizeMultiplier);
```

```
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
decoder(ResourceDecoder<Imagevideowrapper,
GifBitmapwrapper> decoder) {
    super.decoder(decoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
cacheDecoder(ResourceDecoder<File, GifBitmapwrapper>
cacheDecoder) {
    super.cacheDecoder(cacheDecoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
encoder(ResourceEncoder<GifBitmapwrapper> encoder) {
    super.encoder(encoder);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
priority(Priority priority) {
    super.priority(priority);
    return this;
}

public DrawableRequestBuilder<ModelType>
transform(BitmapTransformation... transformations) {
    return bitmapTransform(transformations);
}
```

```
public DrawableRequestBuilder<ModelType> centerCrop()
{
    return transform(glide.getDrawableCenterCrop());
}

public DrawableRequestBuilder<ModelType> fitCenter()
{
    return transform(glide.getDrawableFitCenter());
}

public DrawableRequestBuilder<ModelType>
bitmapTransform(Transformation<Bitmap>...
bitmapTransformations) {
    GifBitmapWrapperTransformation[] transformations
=
    new
GifBitmapWrapperTransformation[bitmapTransformations.length];
    for (int i = 0; i < bitmapTransformations.length;
i++) {
        transformations[i] = new
GifBitmapWrapperTransformation(glide.getBitmapPool(),
bitmapTransformations[i]);
    }
    return transform(transformations);
}

@Override
public DrawableRequestBuilder<ModelType>
transform(Transformation<GifBitmapWrapper>...
transformation) {
    super.transform(transformation);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> transcoder(
```

```
        ResourceTranscoder<GifBitmapWrapper,  
        GlideDrawable> transcoder) {  
            super.transcoder(transcoder);  
            return this;  
        }  
  
        public final DrawableRequestBuilder<ModelType>  
crossFade() {  
            super.animate(new  
DrawableCrossFadeFactory<GlideDrawable>());  
            return this;  
        }  
  
        public DrawableRequestBuilder<ModelType>  
crossFade(int duration) {  
            super.animate(new  
DrawableCrossFadeFactory<GlideDrawable>(duration));  
            return this;  
        }  
  
        public DrawableRequestBuilder<ModelType>  
crossFade(int animationId, int duration) {  
            super.animate(new  
DrawableCrossFadeFactory<GlideDrawable>(context,  
animationId,  
duration));  
            return this;  
        }  
  
        @Override  
        public DrawableRequestBuilder<ModelType>  
dontAnimate() {  
            super.dontAnimate();  
            return this;  
        }  
  
        @Override
```

```
public DrawableRequestBuilder<ModelType>
animate(ViewPropertyAnimation.Animator animator) {
    super.animate(animator);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> animate(int
animationId) {
    super.animate(animationId);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
placeholder(int resourceId) {
    super.placeholder(resourceId);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
placeholder(Drawable drawable) {
    super.placeholder(drawable);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType>
fallback(Drawable drawable) {
    super.fallback(drawable);
    return this;
}

@Override
public DrawableRequestBuilder<ModelType> fallback(int
resourceId) {
    super.fallback(resourceId);
```

```
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> error(int
resourceId) {
        super.error(resourceId);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
error(Drawable drawable) {
        super.error(drawable);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> listener(
        RequestListener<? super ModelType,
GlideDrawable> requestListener) {
        super.listener(requestListener);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
diskCacheStrategy(DiskCacheStrategy strategy) {
        super.diskCacheStrategy(strategy);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
skipMemoryCache(boolean skip) {
        super.skipMemoryCache(skip);
        return this;
    }
```

```
    @Override
    public DrawableRequestBuilder<ModelType> override(int
width, int height) {
        super.override(width, height);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
sourceEncoder(Encoder<Imagevideowrapper> sourceEncoder) {
        super.sourceEncoder(sourceEncoder);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
dontTransform() {
        super.dontTransform();
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
signature(Key signature) {
        super.signature(signature);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType>
load(ModelType model) {
        super.load(model);
        return this;
    }

    @Override
    public DrawableRequestBuilder<ModelType> clone() {
```

```
        return (DrawableRequestBuilder<ModelType>)
super.clone();
    }

@Override
public Target<GlideDrawable> into(ImageView view) {
    return super.into(view);
}

@Override
void applyFitCenter() {
    fitCenter();
}

@Override
void applyCenterCrop() {
    centerCrop();
}
}
```

DrawableRequestBuilder中有很多个方法，这些方法其实就是Glide绝大多数的API了。里面有不少我们在上篇文章中已经用过了，比如说placeholder()方法、error()方法、diskCacheStrategy()方法、override()方法等。当然还有很多暂时还没用到的API，我们会在后面的文章当中学习。

到这里，第二步load()方法也就分析结束了。为什么呢？因为你会发现DrawableRequestBuilder类中有一个into()方法（上述代码第220行），也就是说，最终load()方法返回的其实就是一个DrawableTypeRequest对象。那么接下来我们就要进行第三步了，分析into()方法中的逻辑。

3. into()

如果说前面两步都是在准备开胃小菜的话，那么现在终于要进入主菜了，因为into()方法也是整个Glide图片加载流程中逻辑最复杂的地方。

不过从刚才的代码来看，into()方法中并没有任何逻辑，只有一句super.into(view)。那么很显然，into()方法的具体逻辑都是在DrawableRequestBuilder的父类当中了。

DrawableRequestBuilder的父类是GenericRequestBuilder，我们来看一下GenericRequestBuilder类中的into()方法，如下所示：

```
public Target<TranscodeType> into(ImageView view) {  
    Util.assertMainThread();  
    if (view == null) {  
        throw new IllegalArgumentException("You must pass  
in a non null view");  
    }  
    if (!isTransformationSet && view.getScaleType() !=  
null) {  
        switch (view.getScaleType()) {  
            case CENTER_CROP:  
                applyCenterCrop();  
                break;  
            case FIT_CENTER:  
            case FIT_START:  
            case FIT_END:  
                applyFitCenter();  
                break;  
            // $CASES-OMITTED$  
            default:  
                // Do nothing.  
        }  
    }  
    return into(glide.buildImageViewTarget(view,  
transcodeClass));  
}
```

这里前面一大堆的判断逻辑我们都可以先不用管，等到后面文章讲transform的时候会再进行解释，现在我们只需要关注最后一行代码。最后一行代码先是调用了glide.buildImageViewTarget()方法，这个方法会构建出一个Target对象，Target对象则是用来最终展示图片用的，如果我们跟进去的话会看到如下代码：

```
<R> Target<R> buildImageViewTarget(ImageView imageview,
Class<R> transcodedClass) {
    return imageViewTargetFactory.buildTarget(imageview,
transcodedClass);
}
```

这里其实又是调用了ImageViewTargetFactory的buildTarget()方法，我们继续跟进去，代码如下所示：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view,
Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz))
{
            return (Target<Z>) new
GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new
BitmapImageViewTarget(view);
        } else if
(Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new
DrawableImageViewTarget(view);
        } else {
            throw new IllegalArgumentException("Unhandled
class: " + clazz
                    + ", try .as*
(clazz).transcode(ResourceTranscoder)");
        }
    }
}
```

可以看到，在buildTarget()方法中会根据传入的class参数来构建不同的Target对象。那如果你要分析这个class参数是从哪儿传过来的，这可有得你分析了，简单起见我直接帮大家梳理清楚。这个class参数其实基本上只有两种情况，如果你在使用Glide加载图片的时候调用了asBitmap()方法，那么这里就会构建出BitmapImageViewTarget对象，否则的话构建的都是GlideDrawableImageViewTarget对象。至于上述代码中的DrawableImageViewTarget对象，这个通常都是用不到的，我们可以暂时不用管它。

也就是说，通过glide.buildImageViewTarget()方法，我们构建出了一个GlideDrawableImageViewTarget对象。那现在回到刚才into()方法的最后一行，可以看到，这里又将这个参数传入到了GenericRequestBuilder另一个接收Target对象的into()方法当中了。我们来看一下这个into()方法的源码：

```
public <Y extends Target<TranscodeType>> Y into(Y target)
{
    util.assertMainThread();
    if (target == null) {
        throw new IllegalArgumentException("You must pass
in a non null Target");
    }
    if (!isModelSet) {
        throw new IllegalArgumentException("You must
first set a model (try #load())");
    }
    Request previous = target.getRequest();
    if (previous != null) {
        previous.clear();
        requestTracker.removeRequest(previous);
        previous.recycle();
    }
    Request request = buildRequest(target);
    target.setRequest(request);
    lifecycle.addListener(target);
    requestTracker.runRequest(request);
    return target;
}
```

```
}
```

这里我们还是只抓核心代码，其实只有两行是最关键的，第15行调用 buildRequest()方法构建出了一个Request对象，还有第18行来执行这个 Request。

Request是用来发出加载图片请求的，它是Glide中非常关键的一个组件。我们先来看buildRequest()方法是如何构建Request对象的：

```
private Request buildRequest(Target<TranscodeType> target) {
    if (priority == null) {
        priority = Priority.NORMAL;
    }
    return buildRequestRecursive(target, null);
}

private Request
buildRequestRecursive(Target<TranscodeType> target,
ThumbnailRequestCoordinator parentCoordinator) {
    if (thumbnailRequestBuilder != null) {
        if (isThumbnailBuilt) {
            throw new IllegalStateException("You cannot
use a request as both the main request and a thumbnail, "
+ "consider using clone() on the
request(s) passed to thumbnail()");
        }
        // Recursive case: contains a potentially
        recursive thumbnail request builder.
        if
(thumbnailRequestBuilder.animationFactory.equals(NoAnimat
ion.getFactory())) {
            thumbnailRequestBuilder.animationFactory =
animationFactory;
        }

        if (thumbnailRequestBuilder.priority == null) {
```

```
        thumbnailRequestBuilder.priority =
getThumbnailPriority();
    }

    if (util.isValidDimensions(overrideWidth,
overrideHeight)
    &&
!util.isValidDimensions(thumbnailRequestBuilder.overrideWidth,
overrideHeight,
overrideHeight)) {
    thumbnailRequestBuilder.override(overrideWidth,
overrideHeight);
}

ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
Request fullRequest = obtainRequest(target,
sizeMultiplier, priority, coordinator);
// Guard against infinite recursion.
isThumbnailBuilt = true;
// Recursively generate thumbnail requests.
Request thumbRequest =
thumbnailRequestBuilder.buildRequestRecursive(target,
coordinator);
isThumbnailBuilt = false;
coordinator.setRequests(fullRequest,
thumbRequest);

return coordinator;
} else if (thumbSizeMultiplier != null) {
    // Base case: thumbnail multiplier generates a
    // thumbnail request, but cannot recurse.
    ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
    Request fullRequest = obtainRequest(target,
sizeMultiplier, priority, coordinator);
```

```
        Request thumbnailRequest = obtainRequest(target,
thumbSizeMultiplier, getThumbnailPriority(),
coordinator);
        coordinator.setRequests(fullRequest,
thumbnailRequest);
        return coordinator;
    } else {
        // Base case: no thumbnail.
        return obtainRequest(target, sizeMultiplier,
priority, parentCoordinator);
    }
}

private Request obtainRequest(Target<TranscodeType>
target, float sizeMultiplier, Priority priority,
RequestCoordinator requestCoordinator) {
    return GenericRequest.obtain(
        loadProvider,
        model,
        signature,
        context,
        priority,
        target,
        sizeMultiplier,
        placeholderDrawable,
        placeholderId,
        errorPlaceholder,
        errorId,
        fallbackDrawable,
        fallbackResource,
        requestListener,
        requestCoordinator,
        glide.getEngine(),
        transformation,
        transcodeClass,
        isCacheable,
        animationFactory,
        overridewidth,
```

```
        overrideHeight,  
        diskCacheStrategy);  
    }  
}
```

可以看到，`buildRequest()`方法的内部其实又调用了`buildRequestRecursive()`方法，而`buildRequestRecursive()`方法中的代码虽然有点长，但是其中90%的代码都是在处理缩略图的。如果我们只追主线流程的话，那么只需要看第47行代码就可以了。这里调用了`obtainRequest()`方法来获取一个`Request`对象，而`obtainRequest()`方法中又去调用了`GenericRequest`的`obtain()`方法。注意这个`obtain()`方法需要传入非常多的参数，而其中很多的参数我们都是比较熟悉的，像什么`placeholderId`、`errorPlaceholder`、`diskCacheStrategy`等等。因此，我们就有理由猜测，刚才在`load()`方法中调用的所有API，其实都是在这里组装到`Request`对象当中的。那么我们进入到这个`GenericRequest`的`obtain()`方法瞧一瞧：

```
public final class GenericRequest<A, T, Z, R> implements  
Request, SizeReadyCallback,  
ResourceCallback {  
  
    ...  
  
    public static <A, T, Z, R> GenericRequest<A, T, Z, R>  
obtain(  
        LoadProvider<A, T, Z, R> loadProvider,  
        A model,  
        Key signature,  
        Context context,  
        Priority priority,  
        Target<R> target,  
        float sizeMultiplier,  
        Drawable placeholderDrawable,  
        int placeholderResourceId,  
        Drawable errorDrawable,  
        int errorResourceId,  
        Drawable fallbackDrawable,
```

```
    int fallbackResourceId,
    RequestListener<? super A, R>
requestListener,
    RequestCoordinator requestCoordinator,
    Engine engine,
    Transformation<Z> transformation,
    Class<R> transcodeClass,
    boolean isMemoryCacheable,
    GlideAnimationFactory<R> animationFactory,
    int overridewidth,
    int overrideHeight,
    DiskCacheStrategy diskCacheStrategy) {
    @SuppressWarnings("unchecked")
    GenericRequest<A, T, Z, R> request =
(GenericRequest<A, T, Z, R>) REQUEST_POOL.poll();
    if (request == null) {
        request = new GenericRequest<A, T, Z, R>();
    }
    request.init(loadProvider,
        model,
        signature,
        context,
        priority,
        target,
        sizeMultiplier,
        placeholderDrawable,
        placeholderResourceId,
        errorDrawable,
        errorResourceId,
        fallbackDrawable,
        fallbackResourceId,
        requestListener,
        requestCoordinator,
        engine,
        transformation,
        transcodeClass,
        isMemoryCacheable,
        animationFactory,
```

```
        overrideWidth,  
        overrideHeight,  
        diskCacheStrategy);  
    return request;  
}  
  
...  
}
```

可以看到，这里在第33行去new了一个GenericRequest对象，并在最后一行返回，也就是说，obtain()方法实际上获得的就是一个GenericRequest对象。另外这里又在第35行调用了GenericRequest的init()，里面主要就是一些赋值的代码，将传入的这些参数赋值到GenericRequest的成员变量当中，我们就不再跟进去看了。

好，那现在解决了构建Request对象的问题，接下来我们看一下这个Request对象又是怎么执行的。回到刚才的into()方法，你会发现在第18行调用了requestTracker.runRequest()方法来去执行这个Request，那么我们跟进去瞧一瞧，如下所示：

```
/**  
 * Starts tracking the given request.  
 */  
public void runRequest(Request request) {  
    requests.add(request);  
    if (!isPaused) {  
        request.begin();  
    } else {  
        pendingRequests.add(request);  
    }  
}
```

这里有一个简单的逻辑判断，就是先判断Glide当前是不是处理暂停状态，如果不是暂停状态就调用Request的begin()方法来执行Request，否则的话就先将Request添加到待执行队列里面，等暂停状态解除了之后再执行。

暂停请求的功能仍然不是这篇文章所关心的，这里就直接忽略了，我们重点来看这个begin()方法。由于当前的Request对象是一个GenericRequest，因此这里就需要看GenericRequest中的begin()方法了，如下所示：

```
@Override  
public void begin() {  
    startTime = LogTime.getLogTime();  
    if (model == null) {  
        onException(null);  
        return;  
    }  
    status = Status.WAITING_FOR_SIZE;  
    if (util.isValidDimensions(overrideWidth,  
    overrideHeight)) {  
        onSizeReady(overrideWidth, overrideHeight);  
    } else {  
        target.getSize(this);  
    }  
    if (!isComplete() && !isFailed() &&  
    canNotifyStatusChanged()) {  
        target.onLoadStarted(getPlaceholderDrawable());  
    }  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("finished run method in " +  
        LogTime.getElapsedMillis(startTime));  
    }  
}
```

这里我们来注意几个细节，首先如果model等于null，model也就是我们在第二步load()方法中传入的图片URL地址，这个时候会调用onException()方法。如果你跟到onException()方法里面去看看，你会发现它最终会调用到一个setErrorPlaceholder()当中，如下所示：

```
private void setErrorPlaceholder(Exception e) {  
    if (!canNotifyStatusChanged()) {  
        return;
```

```
        }

        Drawable error = model == null ?
getFallbackDrawable() : null;
        if (error == null) {
            error = getErrorDrawable();
        }
        if (error == null) {
            error = getPlaceholderDrawable();
        }
        target.onLoadFailed(e, error);
    }
}
```

这个方法中会先去获取一个error的占位图，如果获取不到的话会再去获取一个loading占位图，然后调用target.onLoadFailed()方法并将占位图传入。那么onLoadFailed()方法中做了什么呢？我们看一下：

```
public abstract class ImageViewTarget<Z> extends
viewTarget<ImageView, Z> implements
GlideAnimation.ViewAdapter {

    ...

    @Override
    public void onLoadStarted(Drawable placeholder) {
        view.setImageDrawable(placeholder);
    }

    @Override
    public void onLoadFailed(Exception e, Drawable
errorDrawable) {
        view.setImageDrawable(errorDrawable);
    }

    ...
}
```

很简单，其实就是将这张error占位图显示到ImageView上而已，因为现在出现了异常，没办法展示正常的图片了。而如果你仔细看下刚才begin()方法的第15行，你会发现它又调用了一个target.onLoadStarted()方法，并传入了一个loading占位图，在也就说，在图片请求开始之前，会先使用这张占位图代替最终的图片显示。这也是我们在上一篇文章中学过的placeholder()和error()这两个占位图API底层的实现原理。

好，那么我们继续回到begin()方法。刚才讲了占位图的实现，那么具体的图片加载又是从哪里开始的呢？是在begin()方法的第10行和第12行。这里要分两种情况，一种是你使用了override() API为图片指定了一个固定的宽高，一种是没有指定。如果指定了的话，就会执行第10行代码，调用onSizeReady()方法。如果没指定的话，就会执行第12行代码，调用target.getSize()方法。这个target.getSize()方法的内部会根据ImageView的layout_width和layout_height值做一系列的计算，来算出图片应该的宽高。具体的计算细节我就不带着大家分析了，总之在计算完之后，它也会调用onSizeReady()方法。也就是说，不管是哪种情况，最终都会调用到onSizeReady()方法，在这里进行下一步操作。那么我们跟到这个方法里面来：

```
@Override  
public void onSizeReady(int width, int height) {  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("Got onSizeReady in " +  
LogTime.getElapsedMillis(startTime));  
    }  
    if (status != Status.WAITING_FOR_SIZE) {  
        return;  
    }  
    status = Status.RUNNING;  
    width = Math.round(sizeMultiplier * width);  
    height = Math.round(sizeMultiplier * height);  
    ModelLoader<A, T> modelLoader =  
loadProvider.getModelLoader();  
    final DataFetcher<T> dataFetcher =  
modelLoader.getResourceFetcher(model, width, height);  
    if (dataFetcher == null) {
```

```
        onException(new Exception("Failed to load model:  
\" + model + "\"));  
        return;  
    }  
    ResourceTranscoder<Z, R> transcoder =  
loadProvider.getTranscoder();  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("finished setup for calling load in " +  
LogTime.getElapsedMillis(startTime));  
    }  
    loadedFromMemoryCache = true;  
    loadStatus = engine.load(signature, width, height,  
dataFetcher, loadProvider, transformation, transcoder,  
            priority, isMemoryCacheable,  
diskCacheStrategy, this);  
    loadedFromMemoryCache = resource != null;  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logV("finished onSizeReady in " +  
LogTime.getElapsedMillis(startTime));  
    }  
}
```

从这里开始，真正复杂的地方来了，我们需要慢慢进行分析。先来看一下，在第12行调用了loadProvider.getModelLoader()方法，那么我们第一个要搞清楚的就是，这个loadProvider是什么？要搞清楚这点，需要先回到第二步的load()方法当中。还记得load()方法是返回一个DrawableTypeRequest对象吗？刚才我们只是分析了DrawableTypeRequest当中的asBitmap()和asGif()方法，并没有仔细看它的构造函数，现在我们重新来看一下DrawableTypeRequest类的构造函数：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType> implements  
DownloadOptions {
```

```
    private final ModelLoader<ModelType, InputStream>
streamModelLoader;
    private final ModelLoader<ModelType,
ParcelFileDescriptor> fileDescriptorModelLoader;
    private final RequestManager.OptionsApplier
optionsApplier;

    private static <A, Z, R> FixedLoadProvider<A,
ImageVideoWrapper, Z, R> buildProvider(Glide glide,
                                         ModelLoader<A, InputStream>
streamModelLoader,
                                         ModelLoader<A, ParcelFileDescriptor>
fileDescriptorModelLoader, Class<Z> resourceClass,
                                         Class<R> transcodedClass,
                                         ResourceTranscoder<Z, R> transcoder) {
        if (streamModelLoader == null &&
fileDescriptorModelLoader == null) {
            return null;
        }
        if (transcoder == null) {
            transcoder =
glide.buildTranscoder(resourceClass, transcodedClass);
        }
        DataLoadProvider<ImageVideoWrapper, Z>
dataLoadProvider =
glide.buildDataProvider(ImageVideoWrapper.class,
                           resourceClass);
        ImageVideoModelLoader<A> modelLoader = new
ImageVideoModelLoader<A>(streamModelLoader,
                           fileDescriptorModelLoader);
        return new FixedLoadProvider<A,
ImageVideoWrapper, Z, R>(modelLoader, transcoder,
dataLoadProvider);
    }

    DrawableTypeRequest<Class<ModelType> modelClass,
ModelLoader<ModelType, InputStream> streamModelLoader,
```

```
    ModelLoader<ModelType, ParcelFileDescriptor>
fileDescriptorModelLoader, Context context, Glide glide,
    RequestTracker requestTracker, Lifecycle
lifecycle, RequestManager.OptionsApplier optionsApplier)
{
    super(context, modelClass,
        buildProvider(glide, streamModelLoader,
fileDescriptorModelLoader, GifBitmapWrapper.class,
                GlideDrawable.class, null),
        glide, requestTracker, lifecycle);
    this.streamModelLoader = streamModelLoader;
    this.fileDescriptorModelLoader =
fileDescriptorModelLoader;
    this.optionsApplier = optionsApplier;
}
...
}
```

可以看到，这里在第29行，也就是构造函数中，调用了一个buildProvider()方法，并把streamModelLoader和fileDescriptorModelLoader等参数传入到这个方法中，这两个ModelLoader就是之前在loadGeneric()方法中构建出来的。

那么我们再来看一下buildProvider()方法里面做了什么，在第16行调用了glide.buildTranscoder()方法来构建一个ResourceTranscoder，它是用于对图片进行转码的，由于ResourceTranscoder是一个接口，这里实际会构建出一个GifBitmapWrapperDrawableTranscoder对象。

接下来在第18行调用了glide.buildDataProvider()方法来构建一个DataLoadProvider，它是用于对图片进行编解码的，由于DataLoadProvider是一个接口，这里实际会构建出一个ImageVideoGifDrawableLoadProvider对象。

然后在第20行，new了一个ImageVideoModelLoader的实例，并把之前loadGeneric()方法中构建的两个ModelLoader封装到了ImageVideoModelLoader当中。

最后，在第22行，new出一个FixedLoadProvider，并把刚才构建的出来的GifBitmapWrapperDrawableTranscoder、ImageVideoModelLoader、ImageVideoGifDrawableLoadProvider都封装进去，这个也就是onSizeReady()方法中的loadProvider了。

好的，那么我们回到onSizeReady()方法中，在onSizeReady()方法的第12行和第18行，分别调用了loadProvider的getModelLoader()方法和getTranscoder()方法，那么得到的对象也就是刚才我们分析的ImageVideoModelLoader和GifBitmapWrapperDrawableTranscoder了。而在第13行，又调用了ImageVideoModelLoader的getResourceFetcher()方法，这里我们又需要跟进去瞧一瞧了，代码如下所示：

```
public class ImagevideoModelLoader<A> implements ModelLoader<A, Imagevideowrapper> {
    private static final String TAG = "IVML";

    private final ModelLoader<A, InputStream> streamLoader;
    private final ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader;

    public ImagevideoModelLoader(ModelLoader<A, InputStream> streamLoader,
                                 ModelLoader<A, ParcelFileDescriptor> fileDescriptorLoader) {
        if (streamLoader == null && fileDescriptorLoader == null) {
            throw new NullPointerException("At least one of streamLoader and fileDescriptorLoader must be non null");
        }
        this.streamLoader = streamLoader;
        this.fileDescriptorLoader = fileDescriptorLoader;
    }

    @Override
```

```
public DataFetcher<Imagevideowrapper>
getResourceFetcher(A model, int width, int height) {
    DataFetcher<InputStream> streamFetcher = null;
    if (streamLoader != null) {
        streamFetcher =
streamLoader.getResourceFetcher(model, width, height);
    }
    DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher = null;
    if (fileDescriptorLoader != null) {
        fileDescriptorFetcher =
fileDescriptorLoader.getResourceFetcher(model, width,
height);
    }

    if (streamFetcher != null || fileDescriptorFetcher != null) {
        return new ImagevideoFetcher(streamFetcher,
fileDescriptorFetcher);
    } else {
        return null;
    }
}

static class ImagevideoFetcher implements
DataFetcher<Imagevideowrapper> {
    private final DataFetcher<InputStream>
streamFetcher;
    private final DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher;

    public ImagevideoFetcher(DataFetcher<InputStream>
streamFetcher,
                           DataFetcher<ParcelFileDescriptor>
fileDescriptorFetcher) {
        this.streamFetcher = streamFetcher;
        this.fileDescriptorFetcher =
fileDescriptorFetcher;
    }
}
```

```
    }  
    ...  
}  
}
```

可以看到，在第20行会先调用streamLoader.getResourceFetcher()方法获取一个DataFetcher，而这个streamLoader其实就是在loadGeneric()方法中构建出的StreamStringLoader，调用它的getResourceFetcher()方法会得到一个HttpUrlFetcher对象。然后在第28行new出了一个ImageVideoFetcher对象，并把获得的HttpUrlFetcher对象传进去。也就是说，ImageVideoModelLoader的getResourceFetcher()方法得到的是一个ImageVideoFetcher。

那么我们再次回到onSizeReady()方法，在onSizeReady()方法的第23行，这里将刚才获得的ImageVideoFetcher、GifBitmapWrapperDrawableTranscoder等等一系列的值一起传入到了Engine的load()方法当中。接下来我们就要看一看，这个Engine的load()方法当中，到底做了什么？代码如下所示：

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {  
    ...  
    public <T, Z, R> LoadStatus load(Key signature, int  
        width, int height, DataFetcher<T> fetcher,  
        DataLoadProvider<T, Z> loadProvider,  
        Transformation<Z> transformation, ResourceTranscoder<Z,  
        R> transcoder,  
        Priority priority, boolean isMemoryCacheable,  
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)  
{  
    Util.assertMainThread();  
    long startTime = LogTime.getLogTime();
```

```
    final String id = fetcher.getId();
    EngineKey key = keyFactory.buildKey(id,
signature, width, height, loadProvider.getCacheDecoder(),
loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
transcoder,
loadProvider.getSourceEncoder());

    EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);
    if (cached != null) {
        cb.onResourceReady(cached);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            LogWithTimeAndKey("Loaded resource from
cache", startTime, key);
        }
        return null;
    }

    EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
    if (active != null) {
        cb.onResourceReady(active);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            LogWithTimeAndKey("Loaded resource from
active resources", startTime, key);
        }
        return null;
    }

    EngineJob current = jobs.get(key);
    if (current != null) {
        current.addCallback(cb);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            LogWithTimeAndKey("Added to existing
load", startTime, key);
        }
        return new LoadStatus(cb, current);
    }
}
```

```
    }

    EngineJob engineJob = engineJobFactory.build(key,
isMemoryCacheable);

    DecodeJob<T, Z, R> decodeJob = new DecodeJob<T,
Z, R>(key, width, height, fetcher, loadProvider,
transformation,
            transcoder, diskCacheProvider,
diskCacheStrategy, priority);

    EngineRunnable runnable = new
EngineRunnable(engineJob, decodeJob, priority);
    jobs.put(key, engineJob);
    engineJob.addCallback(cb);
    engineJob.start(runnable);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Started new load",
startTime, key);
    }
    return new Loadstatus(cb, engineJob);
}

...
}
```

load()方法中的代码虽然有点长，但大多数的代码都是在处理缓存的。关于缓存的内容我们会在下一篇文章当中学习，现在只需要从第45行看起就行。这里构建了一个EngineJob，它的主要作用就是用来开启线程的，为后面的异步加载图片做准备。接下来第46行创建了一个DecodeJob对象，从名字上来看，它好像是用来对图片进行解码的，但实际上它的任务十分繁重，待会我们就知道了。继续往下看，第48行创建了一个EngineRunnable对象，并且在51行调用了EngineJob的start()方法来运行EngineRunnable对象，这实际上就是让EngineRunnable的run()方法在子线程当中执行了。那么我们现在就可以去看看EngineRunnable的run()方法里做了些什么，如下所示：

```
@Override
public void run() {
    if (isCancelled) {
        return;
    }
    Exception exception = null;
    Resource<?> resource = null;
    try {
        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        exception = e;
    }
    if (isCancelled) {
        if (resource != null) {
            resource.recycle();
        }
        return;
    }
    if (resource == null) {
        onLoadFailed(exception);
    } else {
        onLoadComplete(resource);
    }
}
```

这个方法中的代码并不多，但我们仍然还是要抓重点。在第9行，这里调用了一个decode()方法，并且这个方法返回了一个Resource对象。看上去所有的逻辑应该都在这个decode()方法执行的了，那我们跟进去瞧一瞧：

```
private Resource<?> decode() throws Exception {
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}
```

decode()方法中又分了两种情况，从缓存当中去decode图片的话就会执行decodeFromCache()，否则的话就执行decodeFromSource()。本篇文章中我们不讨论缓存的情况，那么就直接来看decodeFromSource()方法的代码吧，如下所示：

```
private Resource<?> decodeFromSource() throws Exception {
    return decodeJob.decodeFromSource();
}
```

这里又调用了DecodeJob的decodeFromSource()方法。刚才已经说了，DecodeJob的任务十分繁重，我们继续跟进看一看吧：

```
class DecodeJob<A, T, Z> {

    ...

    public Resource<Z> decodeFromSource() throws
Exception {
        Resource<T> decoded = decodeSource();
        return transformEncodeAndTranscode(decoded);
    }

    private Resource<T> decodeSource() throws Exception {
        Resource<T> decoded = null;
        try {
            long startTime = LogTime.getLogTime();
            final A data = fetcher.loadData(priority);
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
```

```
        logWithTimeAndKey("Fetched data",
startTime);
    }
    if (isCancelled) {
        return null;
    }
    decoded = decodeFromSourceData(data);
} finally {
    fetcher.cleanup();
}
return decoded;
}

...
}
```

主要的方法就这些，我都帮大家提取出来了。那么我们先来看一下decodeFromSource()方法，其实它的工作分为两部，第一步是调用decodeSource()方法来获得一个Resource对象，第二步是调用transformEncodeAndTranscode()方法来处理这个Resource对象。

那么我们先来看第一步，decodeSource()方法中的逻辑也并不复杂，首先在第14行调用了fetcher.loadData()方法。那么这个fetcher是什么呢？其实就是刚才在onSizeReady()方法中得到的ImageVideoFetcher对象，这里调用它的 loadData()方法，代码如下所示：

```
@Override
public Imagevideowrapper loadData(Priority priority)
throws Exception {
    InputStream is = null;
    if (streamFetcher != null) {
        try {
            is = streamFetcher.loadData(priority);
        } catch (Exception e) {
            if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(TAG, "Exception fetching input
stream, trying ParcelFileDescriptor", e);
            }
        }
    }
    return new Imagevideowrapper(is);
}
```

```
        }

        if (fileDescriptorFetcher == null) {
            throw e;
        }
    }

}

ParcelFileDescriptor fileDescriptor = null;
if (fileDescriptorFetcher != null) {
    try {
        fileDescriptor =
fileDescriptorFetcher.loadData(priority);
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception fetching
ParcelFileDescriptor", e);
        }
        if (is == null) {
            throw e;
        }
    }
}

return new Imagevideowrapper(is, fileDescriptor);
}
```

可以看到，在ImageVideoFetcher的loadData()方法的第6行，这里又去调用了streamFetcher.loadData()方法，那么这个streamFetcher是什么呢？自然就是刚才在组装ImageVideoFetcher对象时传进来的HttpUrlFetcher了。因此这里又会去调用HttpUrlFetcher的loadData()方法，那么我们继续跟进去瞧一瞧：

```
public class HttpurlFetcher implements
DataFetcher<InputStream> {

    ...

@Override
```

```
public InputStream loadData(Priority priority) throws
Exception {
    return loadDataWithRedirects(glideUrl.toURL(), 0
/*redirects*/, null /*lastUrl*/, glideUrl.getHeaders());
}

private InputStream loadDataWithRedirects(URL url,
int redirects, URL lastUrl, Map<String, String> headers)
throws IOException {
    if (redirects >= MAXIMUM_REDIRECTS) {
        throw new IOException("Too many (> " +
MAXIMUM_REDIRECTS + ") redirects!");
    } else {
        // Comparing the URLs using .equals performs
        // additional network I/O and is generally broken.
        // See
http://michaelscharf.blogspot.com/2006/11/javaneturlequals-and-hashcode-make.html.
        try {
            if (lastUrl != null &&
url.toURI().equals(lastUrl.toURI())) {
                throw new IOException("In re-direct
loop");
            }
        } catch (URISyntaxException e) {
            // Do nothing, this is best effort.
        }
    }
    urlConnection = connectionFactory.build(url);
    for (Map.Entry<String, String> headerEntry :
headers.entrySet()) {

urlConnection.addRequestProperty(headerEntry.getKey(),
headerEntry.getValue());
    }
    urlConnection.setConnectTimeout(2500);
    urlConnection.setReadTimeout(2500);
    urlConnection.setUseCaches(false);
}
```

```
urlConnection.setDoInput(true);

// Connect explicitly to avoid errors in decoders
if connection fails.
    urlConnection.connect();
    if (isCancelled) {
        return null;
    }
    final int statusCode =
urlConnection.getResponseCode();
    if (statusCode / 100 == 2) {
        return
getStreamForSuccessfulRequest(urlConnection);
    } else if (statusCode / 100 == 3) {
        String redirect urlString =
urlConnection.getHeaderField("Location");
        if (TextUtils.isEmpty(redirect urlString)) {
            throw new IOException("Received empty or
null redirect url");
        }
        URL redirectUrl = new URL(url,
redirect urlString);
        return loadDataWithRedirects(redirectUrl,
redirects + 1, url, headers);
    } else {
        if (statusCode == -1) {
            throw new IOException("Unable to retrieve
response code from HttpURLConnection.");
        }
        throw new IOException("Request failed " +
statusCode + ": " + urlConnection.getResponseMessage());
    }
}

private InputStream
getStreamForSuccessfulRequest(HttpURLConnection
urlConnection)
    throws IOException {
```

```
    if  
        (TextUtils.isEmpty(urlConnection.getContentEncoding())) {  
            int contentLength =  
                urlConnection.getContentLength();  
            stream =  
                ContentLengthInputStream.obtain(urlConnection.getInputStream(), contentLength);  
        } else {  
            if (Log.isLoggable(TAG, Log.DEBUG)) {  
                Log.d(TAG, "Got non empty content  
encoding: " + urlConnection.getContentEncoding());  
            }  
            stream = urlConnection.getInputStream();  
        }  
    return stream;  
}  
  
...  
}
```

经过一层一层地跋山涉水，我们终于在这里找到网络通讯的代码了！之前有朋友跟我讲过，说Glide的源码实在是太复杂了，甚至连网络请求是在哪里发出去的都找不到。我们也是经过一段一段又一段的代码跟踪，终于把网络请求的代码给找出来了，实在是太不容易了。

不过也别高兴得太早，现在离最终分析完还早着呢。可以看到，`loadData()`方法只是返回了一个`InputStream`，服务器返回的数据连读都还没开始读呢。所以我们还是要静下心来继续分析，回到刚才`ImageVideoFetcher`的`loadData()`方法中，在这个方法的最后一行，创建了一个`ImageVideoWrapper`对象，并把刚才得到的`InputStream`作为参数传了进去。

然后我们回到再上一层，也就是`DecodeJob`的`decodeSource()`方法当中，在得到了这个`ImageVideoWrapper`对象之后，紧接着又将这个对象传入到了`decodeFromSourceData()`当中，来去解码这个对象。`decodeFromSourceData()`方法的代码如下所示：

```
private Resource<T> decodeFromSourceData(A data) throws  
IOException {  
    final Resource<T> decoded;  
    if (diskCacheStrategy.cacheSource()) {  
        decoded = cacheAndDecodeSourceData(data);  
    } else {  
        long startTime = LogTime.getLogTime();  
        decoded =  
loadProvider.getSourceDecoder().decode(data, width,  
height);  
        if (Log.isLoggable(TAG, Log.VERBOSE)) {  
            logWithTimeAndKey("Decoded from source",  
startTime);  
        }  
    }  
    return decoded;  
}
```

可以看到，这里在第7行调用了
loadProvider.getSourceDecoder().decode()方法来进行解码。
loadProvider就是刚才在onSizeReady()方法中得到的
FixedLoadProvider，而getSourceDecoder()得到的则是一个
GifBitmapWrapperResourceDecoder对象，也就是要调用这个对象的
decode()方法来对图片进行解码。那么我们来看下
GifBitmapWrapperResourceDecoder的代码：

```
public class GifBitmapWrapperResourceDecoder implements  
ResourceDecoder<ImageVideoWrapper, GifBitmapWrapper> {  
  
    ...  
  
    @SuppressWarnings("resource")  
    // @see ResourceDecoder.decode  
    @Override  
    public Resource<GifBitmapWrapper>  
decode(ImageVideoWrapper source, int width, int height)  
throws IOException {
```

```
        ByteArrayPool pool = ByteArrayPool.get();
        byte[] tempBytes = pool.getBytes();
        GifBitmapwrapper wrapper = null;
        try {
            wrapper = decode(source, width, height,
tempBytes);
        } finally {
            pool.releaseBytes(tempBytes);
        }
        return wrapper != null ? new
GifBitmapwrapperResource(wrapper) : null;
    }

    private GifBitmapwrapper decode(Imagevideowrapper
source, int width, int height, byte[] bytes) throws
IOException {
    final GifBitmapwrapper result;
    if (source.getStream() != null) {
        result = decodeStream(source, width, height,
bytes);
    } else {
        result = decodeBitmapwrapper(source, width,
height);
    }
    return result;
}

private GifBitmapwrapper
decodeStream(Imagevideowrapper source, int width, int
height, byte[] bytes)
    throws IOException {
    InputStream bis =
streamFactory.build(source.getStream(), bytes);
    bis.mark(MARK_LIMIT_BYTES);
    ImageHeaderParser.ImageType type =
parser.parse(bis);
    bis.reset();
    GifBitmapwrapper result = null;
```

```
        if (type == ImageHeaderParser.ImageType.GIF) {
            result = decodeGifwrapper(bis, width,
height);
        }
        // Decoding the gif may fail even if the type
matches.

        if (result == null) {
            // we can only reset the buffered
InputStream, so to start from the beginning of the
stream, we need to
                // pass in a new source containing the
buffered stream rather than the original stream.

            Imagevideowrapper forBitmapDecoder = new
Imagevideowrapper(bis, source.getFileDescriptor());
            result =
decodeBitmapwrapper(forBitmapDecoder, width, height);
        }
        return result;
    }

    private GifBitmapwrapper
decodeBitmapwrapper(Imagevideowrapper toDecode, int
width, int height) throws IOException {
    GifBitmapwrapper result = null;
    Resource<Bitmap> bitmapResource =
bitmapDecoder.decode(toDecode, width, height);
    if (bitmapResource != null) {
        result = new GifBitmapwrapper(bitmapResource,
null);
    }
    return result;
}

...
}
```

首先，在decode()方法中，又去调用了另外一个decode()方法的重载。然后在第23行调用了decodeStream()方法，准备从服务器返回的流当中读取数据。decodeStream()方法中会先从流中读取2个字节的数据，来判断这张图是GIF图还是普通的静图，如果是GIF图就调用decodeGifWrapper()方法来进行解码，如果是普通的静图就用调用decodeBitmapWrapper()方法来进行解码。这里我们只分析普通静图的实现流程，GIF图的实现有点过于复杂了，无法在本篇文章当中分析。

然后我们来看一下decodeBitmapWrapper()方法，这里在第52行调用了bitmapDecoder.decode()方法。这个bitmapDecoder是一个ImageVideoBitmapDecoder对象，那么我们来看一下它的代码，如下所示：

```
public class ImagevideoBitmapDecoder implements
ResourceDecoder<ImageVideoWrapper, Bitmap> {
    private final ResourceDecoder<InputStream, Bitmap>
streamDecoder;
    private final ResourceDecoder<ParcelFileDescriptor,
Bitmap> fileDescriptorDecoder;

    public
ImagevideoBitmapDecoder(ResourceDecoder<InputStream,
Bitmap> streamDecoder,
                    ResourceDecoder<ParcelFileDescriptor, Bitmap>
fileDescriptorDecoder) {
        this.streamDecoder = streamDecoder;
        this.fileDescriptorDecoder =
fileDescriptorDecoder;
    }

    @Override
    public Resource<Bitmap> decode(ImageVideoWrapper
source, int width, int height) throws IOException {
        Resource<Bitmap> result = null;
        InputStream is = source.getStream();
        if (is != null) {
            try {
```

```
        result = streamDecoder.decode(is, width,
height);
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Failed to load image from
stream, trying FileDescriptor", e);
        }
    }
    if (result == null) {
        ParcelFileDescriptor fileDescriptor =
source.getFileDescriptor();
        if (fileDescriptor != null) {
            result =
fileDescriptorDecoder.decode(fileDescriptor, width,
height);
        }
    }
    return result;
}
...
}
```

代码并不复杂，在第14行先调用了source.getStream()来获取到服务器返回的InputStream，然后在第17行调用streamDecoder.decode()方法进行解码。streamDecode是一个StreamBitmapDecoder对象，那么我们再来看这个类的源码，如下所示：

```
public class StreamBitmapDecoder implements
ResourceDecoder<InputStream, Bitmap> {

    ...
    private final Downampler downampler;
    private BitmapPool bitmapPool;
    private DecodeFormat decodeFormat;
```

```
public StreamBitmapDecoder(Downsampler downsample, BitmapPool bitmapPool, DecodeFormat decodeFormat) {  
    this.downsample = downsample;  
    this.bitmapPool = bitmapPool;  
    this.decodeFormat = decodeFormat;  
}  
  
@Override  
public Resource<Bitmap> decode(InputStream source, int width, int height) {  
    Bitmap bitmap = downsample.decode(source, bitmapPool, width, height, decodeFormat);  
    return BitmapResource.obtain(bitmap, bitmapPool);  
}  
  
...  
}
```

可以看到，它的decode()方法又去调用了Downsampler的decode()方法。接下来又到了激动人心的时刻了，Downsampler的代码如下所示：

```
public abstract class Downsampler implements  
BitmapDecoder<InputStream> {  
  
...  
  
@Override  
public Bitmap decode(InputStream is, BitmapPool pool, int outwidth, int outHeight, DecodeFormat decodeFormat) {  
    final ByteArrayPool byteArrayPool =  
    ByteArrayPool.get();  
    final byte[] bytesForOptions =  
    byteArrayPool.getBytes();  
    final byte[] bytesForStream =  
    byteArrayPool.getBytes();
```

```
    final BitmapFactory.Options options =
getDefaults();
        // Use to fix the mark limit to avoid allocating
buffers that fit entire images.
    RecyclableBufferedInputStream bufferedStream =
new RecyclableBufferedInputStream(
            is, bytesForStream);
        // Use to retrieve exceptions thrown while
reading.
        // TODO(#126): when the framework no longer
returns partially decoded Bitmaps or provides a way to
determine
        // if a Bitmap is partially decoded, consider
removing.
    ExceptionCatchingInputStream exceptionStream =
ExceptionCatchingInputStream.obtain(bufferedStream);
        // Use to read data.
        // Ensures that we can always reset after reading
an image header so that we can still attempt to decode
the
        // full image even when the header decode fails
and/or overflows our read buffer. See #283.
    MarkEnforcingInputStream invalidatingStream = new
MarkEnforcingInputStream(exceptionStream);
    try {
        exceptionStream.mark(MARK_POSITION);
        int orientation = 0;
        try {
            orientation = new
ImageHeaderParser(exceptionStream).getOrientation();
        } catch (IOException e) {
            if (Log.isLoggable(TAG, Log.WARN)) {
                Log.w(TAG, "Cannot determine the
image orientation from header", e);
            }
        } finally {
            try {
```

```
        exceptionStream.reset();
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.WARN)) {
            Log.w(TAG, "Cannot reset the
input stream", e);
        }
    }
}
options.inTempStorage = bytesForOptions;
final int[] inDimens =
getDimensions(invalidatingStream, bufferedStream,
options);
final int inwidth = inDimens[0];
final int inHeight = inDimens[1];
final int degreesToRotate =
TransformationUtils.getExifOrientationDegrees(orientation
);
final int sampleSize =
getRoundedSampleSize(degreesToRotate, inwidth, inHeight,
outwidth, outHeight);
final Bitmap downsampled =
downsampleWithSize(invalidatingStream, bufferedStream,
options, pool, inwidth, inHeight, sampleSize,
decodeFormat);
// BitmapFactory swallows exceptions during
decodes and in some cases when inBitmap is non null, may
catch
// and log a stack trace but still return a
non null bitmap. To avoid displaying partially decoded
bitmaps,
// we catch exceptions reading from the
stream in our ExceptionCatchingInputStream and throw them
here.
final Exception streamException =
exceptionStream.getException();
if (streamException != null) {
```

```
        throw new
RuntimeException(streamException);
    }
    Bitmap rotated = null;
    if (downsampled != null) {
        rotated =
TransformationUtils.rotateImageExif(downsampled, pool,
orientation);
        if (!downsampled.equals(rotated) &&
!pool.put(downsampled)) {
            downsampled.recycle();
        }
    }
    return rotated;
} finally {
    byteArrayPool.releaseBytes(bytesForOptions);
    byteArrayPool.releaseBytes(bytesForStream);
    exceptionStream.release();
    releaseOptions(options);
}
}

private Bitmap
downsamplewithSize(MarkEnforcingInputStream is,
RecyclableBufferedInputStream bufferedStream,
BitmapFactory.Options options, BitmapPool
pool, int inwidth, int inHeight, int sampleSize,
DecodeFormat decodeFormat) {
    // Prior to KitKat, the inBitmap size must
    // exactly match the size of the bitmap we're decoding.
    Bitmap.Config config = getConfig(is,
decodeFormat);
    options.inSampleSize = sampleSize;
    options.inPreferredConfig = config;
    if ((options.inSampleSize == 1 ||
Build.VERSION_CODES.KITKAT <= Build.VERSION.SDK_INT) &&
shouldUsePool(is)) {
```

```
        int targetwidth = (int) Math.ceil(inwidth /
(double) samplesize);
        int targetHeight = (int) Math.ceil(inHeight /
(double) samplesize);
        // BitmapFactory will clear out the Bitmap
before writing to it, so getDirty is safe.
        setInBitmap(options,
pool.getDirty(targetwidth, targetHeight, config));
    }
    return decodeStream(is, bufferedStream, options);
}

/**
 * A method for getting the dimensions of an image
from the given InputStream.
 *
 * @param is The InputStream representing the image.
 * @param options The options to pass to
 *               {@link
BitmapFactory#decodeStream(InputStream,
android.graphics.Rect,
*               BitmapFactory.Options)}.
 * @return an array containing the dimensions of the
image in the form {width, height}.
 */
public int[] getDimensions(MarkEnforcingInputStream
is, RecyclableBufferedInputStream bufferedStream,
    BitmapFactory.Options options) {
    options.inJustDecodeBounds = true;
    decodeStream(is, bufferedStream, options);
    options.inJustDecodeBounds = false;
    return new int[] { options.outwidth,
options.outHeight };
}

private static Bitmap
decodeStream(MarkEnforcingInputStream is,
RecyclableBufferedInputStream bufferedStream,
```

```
        BitmapFactory.Options options) {
    if (options.inJustDecodeBounds) {
        // This is large, but jpeg headers are not
        size bounded so we need something large enough to
        minimize
        // the possibility of not being able to fit
        enough of the header in the buffer to get the image size
        so
        // that we don't fail to load images. The
        BufferedInputStream will create a new buffer of 2x the
        // original size each time we use up the
        buffer space without passing the mark so this is a
        maximum
        // bound on the buffer size, not a default.
        Most of the time we won't go past our pre-allocated 16kb.
        is.mark(MARK_POSITION);
    } else {
        // Once we've read the image header, we no
        longer need to allow the buffer to expand in size. To
        avoid
        // unnecessary allocations reading image
        data, we fix the mark limit so that it is no larger than
        our
        // current buffer size here. See issue #225.
        bufferedStream.fixMarkLimit();
    }
    final Bitmap result =
BitmapFactory.decodeStream(is, null, options);
    try {
        if (options.inJustDecodeBounds) {
            is.reset();
        }
    } catch (IOException e) {
        if (Log.isLoggable(TAG, Log.ERROR)) {
            Log.e(TAG, "Exception loading
inDecodeBounds=" + options.inJustDecodeBounds
                    + " sample=" +
options.inSampleSize, e);
    }
}
```

```
        }

    }

    return result;
}

...
}
```

可以看到，对服务器返回的InputStream的读取，以及对图片的加载全都在这里了。当然这里其实处理了很多的逻辑，包括对图片的压缩，甚至还有旋转、圆角等逻辑处理，但是我们目前只需要关注主线逻辑就行了。decode()方法执行之后，会返回一个Bitmap对象，那么图片在这里其实也就已经被加载出来了，剩下的工作就是如果让这个Bitmap显示到界面上，我们继续往下分析。

回到刚才的StreamBitmapDecoder当中，你会发现，它的decode()方法返回的是一个Resource对象。而我们从Downsampler中得到的是一个Bitmap对象，因此这里在第18行又调用了BitmapResource.obtain()方法，将Bitmap对象包装成了Resource对象。代码如下所示：

```
public class BitmapResource implements Resource<Bitmap> {
    private final Bitmap bitmap;
    private final BitmapPool bitmapPool;

    /**
     * Returns a new {@link BitmapResource} wrapping the given {@link Bitmap}
     * if the Bitmap is non-null or null if the
     * given Bitmap is null.
     *
     * @param bitmap A Bitmap.
     * @param bitmapPool A non-null {@link BitmapPool}.
     */
    public static BitmapResource obtain(Bitmap bitmap,
                                       BitmapPool bitmapPool) {
        if (bitmap == null) {
```

```
        return null;
    } else {
        return new BitmapResource(bitmap,
bitmapPool);
    }
}

public BitmapResource(Bitmap bitmap, BitmapPool
bitmapPool) {
    if (bitmap == null) {
        throw new NullPointerException("Bitmap must
not be null");
    }
    if (bitmapPool == null) {
        throw new NullPointerException("BitmapPool
must not be null");
    }
    this.bitmap = bitmap;
    this.bitmapPool = bitmapPool;
}

@Override
public Bitmap get() {
    return bitmap;
}

@Override
public int getSize() {
    return Util.getBitmapBytesize(bitmap);
}

@Override
public void recycle() {
    if (!bitmapPool.put(bitmap)) {
        bitmap.recycle();
    }
}
}
```

BitmapResource的源码也非常简单，经过这样一层包装之后，如果我还需要获取Bitmap，只需要调用Resource的get()方法就可以了。

然后我们需要一层层继续向上返回，StreamBitmapDecoder会将值返回到ImageVideoBitmapDecoder当中，而ImageVideoBitmapDecoder又会将值返回到GifBitmapWrapperResourceDecoder的decodeBitmapWrapper()方法当中。由于代码隔得有点太远了，我重新把decodeBitmapWrapper()方法的代码贴一下：

```
private GifBitmapwrapper  
decodeBitmapwrapper(Imagevideowrapper toDecode, int  
width, int height) throws IOException {  
    GifBitmapwrapper result = null;  
    Resource<Bitmap> bitmapResource =  
    bitmapDecoder.decode(toDecode, width, height);  
    if (bitmapResource != null) {  
        result = new GifBitmapwrapper(bitmapResource,  
null);  
    }  
    return result;  
}
```

可以看到，decodeBitmapWrapper()方法返回的是一个GifBitmapWrapper对象。因此，这里在第5行，又将Resource封装到了一个GifBitmapWrapper对象当中。这个GifBitmapWrapper顾名思义，就是既能封装GIF，又能封装Bitmap，从而保证了不管是什类型的图片Glide都能从容应对。我们顺便来看下GifBitmapWrapper的源码吧，如下所示：

```
public class GifBitmapwrapper {  
    private final Resource<GifDrawable> gifResource;  
    private final Resource<Bitmap> bitmapResource;  
  
    public GifBitmapwrapper(Resource<Bitmap>  
    bitmapResource, Resource<GifDrawable> gifResource) {
```

```
        if (bitmapResource != null && gifResource !=  
null) {  
            throw new IllegalArgumentException("Can only  
contain either a bitmap resource or a gif resource, not  
both");  
        }  
        if (bitmapResource == null && gifResource ==  
null) {  
            throw new IllegalArgumentException("Must  
contain either a bitmap resource or a gif resource");  
        }  
        this.bitmapResource = bitmapResource;  
        this.gifResource = gifResource;  
    }  
  
    /**  
     * Returns the size of the wrapped resource.  
     */  
    public int getSize() {  
        if (bitmapResource != null) {  
            return bitmapResource.getSize();  
        } else {  
            return gifResource.getSize();  
        }  
    }  
  
    /**  
     * Returns the wrapped {@link Bitmap} resource if it  
     * exists, or null.  
     */  
    public Resource<Bitmap> getBitmapResource() {  
        return bitmapResource;  
    }  
  
    /**  
     * Returns the wrapped {@link GifDrawable} resource  
     * if it exists, or null.  
     */
```

```
public Resource<GifDrawable> getGifResource() {  
    return gifResource;  
}  
}
```

还是比较简单的，就是分别对gifResource和bitmapResource做了一层封装而已，相信没有什么解释的必要。

然后这个GifBitmapWrapper对象会一直向上返回，返回到GifBitmapWrapperResourceDecoder最外层的decode()方法的时候，会对它再做一次封装，如下所示：

```
@Override  
public Resource<GifBitmapWrapper>  
decode(Imagevideowrapper source, int width, int height)  
throws IOException {  
    ByteArrayPool pool = ByteArrayPool.get();  
    byte[] tempBytes = pool.getBytes();  
    GifBitmapWrapper wrapper = null;  
    try {  
        wrapper = decode(source, width, height,  
tempBytes);  
    } finally {  
        pool.releaseBytes(tempBytes);  
    }  
    return wrapper != null ? new  
GifBitmapWrapperResource(wrapper) : null;  
}
```

可以看到，这里在第11行，又将GifBitmapWrapper封装到了一个GifBitmapWrapperResource对象当中，最终返回的是一个Resource对象。这个GifBitmapWrapperResource和刚才的BitmapResource是相似的，它们都实现的Resource接口，都可以通过get()方法来获取封装起来的具体内容。GifBitmapWrapperResource的源码如下所示：

```
public class GifBitmapWrapperResource implements
Resource<GifBitmapWrapper> {
    private final GifBitmapWrapper data;

    public GifBitmapWrapperResource(GifBitmapWrapper
data) {
        if (data == null) {
            throw new NullPointerException("Data must not
be null");
        }
        this.data = data;
    }

    @Override
    public GifBitmapWrapper get() {
        return data;
    }

    @Override
    public int getSize() {
        return data.getSize();
    }

    @Override
    public void recycle() {
        Resource<Bitmap> bitmapResource =
data.getBitmapResource();
        if (bitmapResource != null) {
            bitmapResource.recycle();
        }
        Resource<GifDrawable> gifDataSource =
data.getGifResource();
        if (gifDataSource != null) {
            gifDataSource.recycle();
        }
    }
}
```

经过这一层的封装之后，我们从网络上得到的图片就能够以Resource接口的形式返回，并且还能同时处理Bitmap图片和GIF图片这两种情况。

那么现在我们可以回到DecodeJob当中了，它的decodeFromSourceData()方法返回的是一个Resource对象，其实也就是Resource对象了。然后继续向上返回，最终返回到decodeFromSource()方法当中，如下所示：

```
public Resource<Z> decodeFromSource() throws Exception {  
    Resource<T> decoded = decodeSource();  
    return transformEncodeAndTranscode(decoded);  
}
```

刚才我们就是从这里跟进到decodeSource()方法当中，然后执行了一大堆一大堆的逻辑，最终得到了这个Resource对象。然而你会发现，decodeFromSource()方法最终返回的却是一个Resource对象，那么这到底是怎么回事呢？我们就需要跟进到transformEncodeAndTranscode()方法来瞧一瞧了，代码如下所示：

```
private Resource<Z>  
transformEncodeAndTranscode(Resource<T> decoded) {  
    long startTime = LogTime.getLogTime();  
    Resource<T> transformed = transform(decoded);  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logWithTimeAndKey("Transformed resource from  
source", startTime);  
    }  
    writeTransformedToCache(transformed);  
    startTime = LogTime.getLogTime();  
    Resource<Z> result = transcode(transformed);  
    if (Log.isLoggable(TAG, Log.VERBOSE)) {  
        logWithTimeAndKey("Transcoded transformed from  
source", startTime);  
    }  
    return result;  
}
```

```
private Resource<Z> transcode(Resource<T> transformed) {  
    if (transformed == null) {  
        return null;  
    }  
    return transcoder.transcode(transformed);  
}
```

首先，这个方法开头的几行transform还有cache，这都是我们后面才会学习的东西，现在不用管它们就可以了。需要注意的是第9行，这里调用了一个transcode()方法，就把Resource对象转换成Resource对象了。

而transcode()方法中又是调用了transcoder的transcode()方法，那么这个transcoder是什么呢？其实这也是Glide源码特别难懂的原因之一，就是它用到的很多对象都是很早很早之前就初始化的，在初始化的时候你可能完全就没有留意过它，因为一时半会根本就用不着，但是真正需要用到的时候你却早就记不起来这个对象是从哪儿来的了。

那么这里我来提醒一下大家吧，在第二步load()方法返回的那个DrawableTypeRequest对象，它的构建函数中去构建了一个FixedLoadProvider对象，然后我们将三个参数传入到了FixedLoadProvider当中，其中就有一个GifBitmapWrapperDrawableTranscoder对象。后来在onSizeReady()方法中获取到了这个参数，并传递到了Engine当中，然后又由Engine传递到了DecodeJob当中。因此，这里的transcoder其实就是这个GifBitmapWrapperDrawableTranscoder对象。那么我们来看一下它的源码：

```
public class GifBitmapWrapperDrawableTranscoder  
implements ResourceTranscoder<GifBitmapWrapper,  
GlideDrawable> {  
    private final ResourceTranscoder<Bitmap,  
GlideBitmapDrawable> bitmapDrawableResourceTranscoder;  
  
    public GifBitmapWrapperDrawableTranscoder(  
        ResourceTranscoder<Bitmap,  
GlideBitmapDrawable> bitmapDrawableResourceTranscoder) {
```

```
        this.bitmapDrawableResourceTranscoder =
bitmapDrawableResourceTranscoder;
    }

    @Override
    public Resource<GlideDrawable>
transcode(Resource<GifBitmapWrapper> toTranscode) {
    GifBitmapWrapper gifBitmap = toTranscode.get();
    Resource<Bitmap> bitmapResource =
gifBitmap.getBitmapResource();
    final Resource<? extends GlideDrawable> result;
    if (bitmapResource != null) {
        result =
bitmapDrawableResourceTranscoder.transcode(bitmapResource);
    } else {
        result = gifBitmap.getGifResource();
    }
    return (Resource<GlideDrawable>) result;
}

...
}
```

这里我来简单解释一下，`GifBitmapWrapperDrawableTranscoder`的核心作用就是用来转码的。因为`GifBitmapWrapper`是无法直接显示到`ImageView`上面的，只有`Bitmap`或者`Drawable`才能显示到`ImageView`上。因此，这里的`transcode()`方法先从`Resource`中取出`GifBitmapWrapper`对象，然后再从`GifBitmapWrapper`中取出`Resource`对象。

接下来做了一个判断，如果`Resource`为空，那么说明此时加载的是GIF图，直接调用`getGifResource()`方法将图片取出即可，因为Glide用于加载GIF图片是使用的`GifDrawable`这个类，它本身就是一个`Drawable`对象了。而如果`Resource`不为空，那么就需要再做一次转码，将`Bitmap`转换

成Drawable对象才行，因为要保证静图和动图的类型一致性，不然逻辑上是不好处理的。

这里在第15行又进行了一次转码，是调用的GlideBitmapDrawableTranscoder对象的transcode()方法，代码如下所示：

```
public class GlideBitmapDrawableTranscoder implements ResourceTranscoder<Bitmap, GlideBitmapDrawable> {
    private final Resources resources;
    private final BitmapPool bitmapPool;

    public GlideBitmapDrawableTranscoder(Context context) {
        this(context.getResources(),
Glide.get(context).getBitmapPool());
    }

    public GlideBitmapDrawableTranscoder(Resources resources, BitmapPool bitmapPool) {
        this.resources = resources;
        this.bitmapPool = bitmapPool;
    }

    @Override
    public Resource<GlideBitmapDrawable>
transcode(Resource<Bitmap> toTranscode) {
        GlideBitmapDrawable drawable = new
GlideBitmapDrawable(resources, toTranscode.get());
        return new GlideBitmapDrawableResource(drawable,
bitmapPool);
    }

    ...
}
```

可以看到，这里new出了一个GlideBitmapDrawable对象，并把Bitmap封装到里面。然后对GlideBitmapDrawable再进行一次封装，返回一个Resource对象。

现在再返回到GifBitmapWrapperDrawableTranscoder的transcode()方法中，你会发现它们的类型就一致了。因为不管是静图的Resource对象，还是动图的Resource对象，它们都是属于父类Resource对象的。因此transcode()方法也是直接返回了Resource，而这个Resource其实也就是转换过后的Resource了。

那么我们继续回到DecodeJob当中，它的decodeFromSource()方法得到了Resource对象，当然也就是Resource对象。然后继续向上返回会回到EngineRunnable的decodeFromSource()方法，再回到decode()方法，再回到run()方法当中。那么我们重新再贴一下EngineRunnable run()方法的源码：

```
@Override  
public void run() {  
    if (isCancelled) {  
        return;  
    }  
    Exception exception = null;  
    Resource<?> resource = null;  
    try {  
        resource = decode();  
    } catch (Exception e) {  
        if (Log.isLoggable(TAG, Log.VERBOSE)) {  
            Log.v(TAG, "Exception decoding", e);  
        }  
        exception = e;  
    }  
    if (isCancelled) {  
        if (resource != null) {  
            resource.recycle();  
        }  
        return;  
    }  
    if (resource == null) {
```

```
        onLoadFailed(exception);
    } else {
        onLoadComplete(resource);
    }
}
```

也就是说，经过第9行decode()方法的执行，我们最终得到了这个Resource对象，那么接下来就是如何将它显示出来了。可以看到，这里在第25行调用了onLoadComplete()方法，表示图片加载已经完成了，代码如下所示：

```
private void onLoadComplete(Resource resource) {
    manager.onResourceReady(resource);
}
```

这个manager就是EngineJob对象，因此这里实际上调用的是EngineJob的onResourceReady()方法，代码如下所示：

```
class EngineJob implements
EngineRunnable.EngineRunnableManager {

    private static final Handler MAIN_THREAD_HANDLER =
new Handler(Looper.getMainLooper(), new
MainThreadCallback());

    private final List<ResourceCallback> cbs = new
ArrayList<ResourceCallback>();

    ...

    public void addCallback(ResourceCallback cb) {
        Util.assertMainThread();
        if (hasResource) {
            cb.onResourceReady(engineResource);
        } else if (hasException) {
            cb.onException(exception);
        }
    }
}
```

```
        } else {
            cbs.add(cb);
        }
    }

@Override
public void onResourceReady(final Resource<?>
resource) {
    this.resource = resource;
    MAIN_THREAD_HANDLER.obtainMessage(MSG_COMPLETE,
this).sendToTarget();
}

private void handleResultOnMainThread() {
    if (isCancelled) {
        resource.recycle();
        return;
    } else if (cbs.isEmpty()) {
        throw new IllegalStateException("Received a
resource without any callbacks to notify");
    }
    engineResource =
engineResourceFactory.build(resource, isCacheable);
    hasResource = true;
    engineResource.acquire();
    listener.onEngineJobComplete(key,
engineResource);
    for (ResourceCallback cb : cbs) {
        if (!isIgnoredCallbacks(cb)) {
            engineResource.acquire();
            cb.onResourceReady(engineResource);
        }
    }
    engineResource.release();
}

@Override
public void onException(final Exception e) {
```

```
        this.exception = e;
        MAIN_THREAD_HANDLER.obtainMessage(MSG_EXCEPTION,
this).sendToTarget();
    }

private void handleExceptionOnMainThread() {
    if (isCancelled) {
        return;
    } else if (cbs.isEmpty()) {
        throw new IllegalStateException("Received an
exception without any callbacks to notify");
    }
    hasException = true;
    listener.onEngineJobComplete(key, null);
    for (ResourceCallback cb : cbs) {
        if (!isIgnoredCallbacks(cb)) {
            cb.onException(exception);
        }
    }
}

private static class MainThreadCallback implements
Handler.Callback {

    @Override
    public boolean handleMessage(Message message) {
        if (MSG_COMPLETE == message.what ||

MSG_EXCEPTION == message.what) {
            EngineJob job = (EngineJob) message.obj;
            if (MSG_COMPLETE == message.what) {
                job.handleResultOnMainThread();
            } else {
                job.handleExceptionOnMainThread();
            }
            return true;
        }
        return false;
    }
}
```

```
}
```

```
...
```

```
}
```

可以看到，这里在onResourceReady()方法使用Handler发出了一条MSG_COMPLETE消息，那么在MainThreadCallback的handleMessage()方法中就会收到这条消息。从这里开始，所有的逻辑又回到主线程当中进行了，因为很快就需要更新UI了。

然后在第72行调用了handleResultOnMainThread()方法，这个方法中又通过一个循环，调用了所有ResourceCallback的onResourceReady()方法。那么这个ResourceCallback是什么呢？答案在addCallback()方法当中，它会向cbs集合中去添加ResourceCallback。那么这个addCallback()方法又是哪里调用的呢？其实调用的地方我们早就已经看过了，只不过之前没有注意，现在重新来看一下Engine的load()方法，如下所示：

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {  
  
    ...  
  
    public <T, Z, R> LoadStatus load(Key signature, int  
        width, int height, DataFetcher<T> fetcher,  
        DataLoadProvider<T, Z> loadProvider,  
        Transformation<Z> transformation, ResourceTranscoder<Z,  
        R> transcoder, Priority priority,  
        boolean isMemoryCacheable, DiskCacheStrategy  
        diskCacheStrategy, ResourceCallback cb) {  
  
        ...  
  
        EngineJob engineJob = engineJobFactory.build(key,  
            isMemoryCacheable);
```

```
        DecodeJob<T, Z, R> decodeJob = new DecodeJob<T,
Z, R>(key, width, height, fetcher, loadProvider,
transformation,
        transcoder, diskCacheProvider,
diskCacheStrategy, priority);
        EngineRunnable runnable = new
EngineRunnable(engineJob, decodeJob, priority);
        jobs.put(key, engineJob);
        engineJob.addCallback(cb);
        engineJob.start(runnable);

        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Started new load",
startTime, key);
        }
        return new LoadStatus(cb, engineJob);
    }

    ...
}
```

这次把目光放在第18行上面，看到了吗？就是在这里调用的EngineJob的addCallback()方法来注册的一个ResourceCallback。那么接下来的问题就是，Engine.load()方法的ResourceCallback参数又是谁传过来的呢？这就需要回到GenericRequest的onSizeReady()方法当中了，我们看到ResourceCallback是load()方法的最后一个参数，那么在onSizeReady()方法中调用load()方法时传入的最后一个参数是什么？代码如下所示：

```
public final class GenericRequest<A, T, Z, R> implements
Request, SizeReadyCallback,
ResourceCallback {

    ...
    @Override
    public void onSizeReady(int width, int height) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
```

```
        logV("Got onSizeReady in " +
LogTime.getElapsedMillis(startTime));
    }
    if (status != Status.WAITING_FOR_SIZE) {
        return;
    }
    status = Status.RUNNING;
    width = Math.round(sizeMultiplier * width);
    height = Math.round(sizeMultiplier * height);
    ModelLoader<A, T> modelLoader =
loadProvider.getModelLoader();
    final DataFetcher<T> dataFetcher =
modelLoader.getResourceFetcher(model, width, height);
    if (dataFetcher == null) {
        onException(new Exception("Failed to load
model: '" + model + "'"));
        return;
    }
    ResourceTranscoder<Z, R> transcoder =
loadProvider.getTranscoder();
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished setup for calling load in " +
LogTime.getElapsedMillis(startTime));
    }
    loadedFromMemoryCache = true;
    loadStatus = engine.load(signature, width,
height, dataFetcher, loadProvider, transformation,
        transcoder, priority, isMemoryCacheable,
diskCacheStrategy, this);
    loadedFromMemoryCache = resource != null;
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logV("finished onSizeReady in " +
LogTime.getElapsedMillis(startTime));
    }
}

...
}
```

请将目光锁定在第29行的最后一个参数，this。没错，就是this。GenericRequest本身就实现了ResourceCallback的接口，因此EngineJob的回调最终其实就是回调到了GenericRequest的onResourceReady()方法当中了，代码如下所示：

```
public void onResourceReady(Resource<?> resource) {
    if (resource == null) {
        onException(new Exception("Expected to receive a
Resource<R> with an object of " + transcodeClass
                + " inside, but instead got null."));
        return;
    }
    Object received = resource.get();
    if (received == null ||
!transcodeClass.isAssignableFrom(received.getClass())) {
        releaseResource(resource);
        onException(new Exception("Expected to receive an
object of " + transcodeClass
                + " but instead got " + (received != null
? received.getClass() : "") + "{" + received + "}"
                + " inside Resource{" + resource + "}."
                + (received != null ? "" : " "
                + "To indicate failure return a null
Resource object, "
                + "rather than a Resource object
containing null data."))
    );
    return;
}
if (!canSetResource()) {
    releaseResource(resource);
    // We can't set the status to complete before
asking canSetResource().
    status = Status.COMPLETE;
    return;
}
```

```
    onResourceReady(resource, (R) received);
}

private void onResourceReady(Resource<?> resource, R
result) {
    // we must call isFirstReadyResource before setting
    status.
    boolean isFirstResource = isFirstReadyResource();
    status = Status.COMPLETE;
    this.resource = resource;
    if (requestListener == null ||
!requestListener.onResourceReady(result, model, target,
loadedFromMemoryCache,
        isFirstResource)) {
        GlideAnimation<R> animation =
animationFactory.build(loadedFromMemoryCache,
isFirstResource);
        target.onResourceReady(result, animation);
    }
    notifyLoadSuccess();
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        LogV("Resource ready in " +
LogTime.getElapsedMillis(startTime) + " size: "
            + (resource.getSize() * TO_MEGABYTE) + " "
fromCache: " + loadedFromMemoryCache);
    }
}
```

这里有两个onResourceReady()方法，首先在第一个onResourceReady()方法当中，调用resource.get()方法获取到了封装的图片对象，也就是GlideBitmapDrawable对象，或者是GifDrawable对象。然后将这个值传入到了第二个onResourceReady()方法当中，并在第36行调用了target.onResourceReady()方法。

那么这个target又是什么呢？这个又需要向上翻很久了，在第三步into()方法的一开始，我们就分析了在into()方法的最后一行，调用了glide.buildImageViewTarget()方法来构建出一个Target，而这个Target就是一个GlideDrawableImageViewTarget对象。

那么我们去看GlideDrawableImageViewTarget的源码就可以了，如下所示：

```
public class GlideDrawableImageViewTarget extends  
ImageViewTarget<GlideDrawable> {  
    private static final float SQUARE_RATIO_MARGIN =  
0.05f;  
    private int maxLoopCount;  
    private GlideDrawable resource;  
  
    public GlideDrawableImageViewTarget(ImageView view) {  
        this(view, GlideDrawable.LOOP_FOREVER);  
    }  
  
    public GlideDrawableImageViewTarget(ImageView view,  
int maxLoopCount) {  
        super(view);  
        this.maxLoopCount = maxLoopCount;  
    }  
  
    @Override  
    public void onResourceReady(GlideDrawable resource,  
GlideAnimation<? super GlideDrawable> animation) {  
        if (!resource.isAnimated()) {  
            float viewRatio = view.getWidth() / (float)  
view.getHeight();  
            float drawableRatio =  
resource.getIntrinsicWidth() / (float)  
resource.getIntrinsicHeight();  
            if (Math.abs(viewRatio - 1f) <=  
SQUARE_RATIO_MARGIN  
                && Math.abs(drawableRatio - 1f) <=  
SQUARE_RATIO_MARGIN) {
```

```
        resource = new SquaringDrawable(resource,
view.getWidth());
    }
}

super.onResourceReady(resource, animation);
this.resource = resource;
resource.setLoopCount(maxLoopCount);
resource.start();
}

@Override
protected void setResource(GlideDrawable resource) {
    view.setImageDrawable(resource);
}

@Override
public void onStart() {
    if (resource != null) {
        resource.start();
    }
}

@Override
public void onStop() {
    if (resource != null) {
        resource.stop();
    }
}
}
```

在GlideDrawableImageViewTarget的onResourceReady()方法中做了一些逻辑处理，包括如果是GIF图片的话，就调用resource.start()方法开始播放图片，但是好像并没有看到哪里有将GlideDrawable显示到ImageView上的逻辑。

确实没有，不过父类里面有，这里在第25行调用了super.onResourceReady()方法，GlideDrawableImageViewTarget的父类是ImageViewTarget，我们来看下它的代码吧：

```
public abstract class ImageViewTarget<Z> extends  
viewTarget<ImageView, Z> implements  
GlideAnimation.ViewAdapter {  
  
    ...  
  
    @Override  
    public void onResourceReady(Z resource,  
GlideAnimation<? super Z> glideAnimation) {  
        if (glideAnimation == null ||  
!glideAnimation.animate(resource, this)) {  
            setResource(resource);  
        }  
    }  
  
    protected abstract void setResource(Z resource);  
}
```

可以看到，在ImageViewTarget的onResourceReady()方法当中调用了setResource()方法，而ImageViewTarget的setResource()方法是一个抽象方法，具体的实现还是在子类那边实现的。

那子类的setResource()方法是怎么实现的呢？回头再来看一下GlideDrawableImageViewTarget的setResource()方法，没错，调用的view.setImageDrawable()方法，而这个view就是ImageView。代码执行到这里，图片终于也就显示出来了。

那么，我们对Glide执行流程的源码分析，到这里也终于结束了。

总结

真是好长的一篇文章，这也可能是我目前所写过的最长的一篇文章了。如果你之前没有读过Glide的源码，真的很难相信，这短短一行代码：

```
Glide.with(this).load(url).into(imageview);
```

背后竟然蕴藏着如此极其复杂的逻辑吧？

不过Glide也并不是有意要将代码写得如此复杂，实在是因为Glide的功能太强大了，而上述代码只是使用了Glide最最基本的功能而已。

现在通过两篇文章，我们已经掌握了Glide的基本用法，并且通过阅读源码了解了Glide总的执行流程。接下来的几篇文章，我会带大家深入到Glide源码的某一处细节，学习Glide更多的高级使用技巧，感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（三），深入探究Glide的缓存机制](#)。

3.12 Android图片加载框架最全解析（三），深入探究Glide的缓存机制

在本系列的上一篇文章中，我带着大家一起阅读了一遍Glide的源码，初步了解了这个强大的图片加载框架的基本执行流程。

不过，上一篇文章只能说是比较粗略地阅读了Glide整个执行流程方面的源码，搞明白了Glide的基本工作原理，但并没有去深入分析每一处的细节（事实上也不可能在一篇文章中深入分析每一处源码的细节）。那么从本篇文章开始，我们就一篇篇地来针对Glide某一块功能进行深入地分析，慢慢将Glide中的各项功能进行全面掌握。

今天我们就先从缓存这一块内容开始入手吧。不过今天文章中的源码都建在上一篇源码分析的基础之上，还没有看过上一篇文章的朋友，建议先去阅读 [Android图片加载框架最全解析（二），从源码的角度理解Glide的执行流程](#)。

Glide缓存简介

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就分别来分析一下这两种缓存的使用方法以及它们的实现原理。

缓存Key

既然是缓存功能，就必然会有用于进行缓存的Key。那么Glide的缓存Key是怎么生成的呢？我不得不说，Glide的缓存Key生成规则非常繁琐，决定缓存Key的参数竟然有10个之多。不过繁琐归繁琐，至少逻辑还是比较简单的，我们先来看一下Glide缓存Key的生成逻辑。

生成缓存Key的代码在Engine类的load()方法当中，这部分代码我们在上一篇文章当中已经分析过了，只不过当时忽略了缓存相关的内容，那么我们现在重新来看一下：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    public <T, Z, R> LoadStatus load(Key signature, int
        width, int height, DataFetcher<T> fetcher,
        DataLoadProvider<T, Z> loadProvider,
        Transformation<Z> transformation, ResourceTranscoder<Z,
        R> transcoder,
        Priority priority, boolean isMemoryCacheable,
        DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)
    {
        Util.assertMainThread();
        long startTime = LogTime.getLogTime();

        final String id = fetcher.getId();
        EngineKey key = keyFactory.buildKey(id,
            signature, width, height, loadProvider.getCacheDecoder(),
```

```
        loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
transcoder,
loadProvider.getSourceEncoder());
}

...
}

}
```

可以看到，这里在第11行调用了fetcher.getId()方法获得了一个id字符串，这个字符串也就是我们要加载的图片的唯一标识，比如说如果是一张网络上的图片的话，那么这个id就是这张图片的url地址。

接下来在第12行，将这个id连同着signature、width、height等等10个参数一起传入到EngineKeyFactory的buildKey()方法当中，从而构建出了一个EngineKey对象，这个EngineKey也就是Glide中的缓存Key了。

可见，决定缓存Key的条件非常多，即使你用override()方法改变了一下图片的width或者height，也会生成一个完全不同的缓存Key。

EngineKey类的源码大家有兴趣可以自己去看一下，其实主要就是重写了equals()和hashCode()方法，保证只有传入EngineKey的所有参数都相同的情况下才认为是同一个EngineKey对象，我就不在这里将源码贴出来了。

内存缓存

有了缓存Key，接下来就可以开始进行缓存了，那么我们先从内存缓存看起。

首先你要知道，默认情况下，Glide自动就是开启内存缓存的。也就是说，当我们使用Glide加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用Glide再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个RecyclerView当中反复上下滑动，RecyclerView中只要是Glide加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而Glide最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为Glide默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide对此提供了接口：

```
Glide.with(this)
    .load(url)
    .skipMemoryCache(true)
    .into(imageview);
```

可以看到，只需要调用skipMemoryCache()方法并传入true，就表示禁用掉Glide的内存缓存功能。

没错，关于Glide内存缓存的用法就只有这么多，可以说是相当简单。但是我们不可能只停留在这么简单的层面上，接下来就让我们就通过阅读源码来分析一下Glide的内存缓存功能是如何实现的。

其实说到内存缓存的实现，非常容易就让人想到LruCache算法（Least Recently Used），也叫近期最少使用算法。它的主要算法原理就是把最近使用的对象用强引用存储在LinkedHashMap中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。LruCache的用法也比较简单，我在[Android高效加载大图、多图解决方案，有效避免程序OOM](#)这篇文章当中有提到过它的用法，感兴趣的朋友可以去参考一下。

那么不必多说，Glide内存缓存的实现自然也是使用的LruCache算法。不过除了LruCache算法之外，Glide还结合了一种弱引用的机制，共同完成了内存缓存功能，下面就让我们来通过源码分析一下。

首先回忆一下，在上一篇文章的第二步load()方法中，我们当时分析到了在loadGeneric()方法中会调用Glide.buildStreamModelLoader()方法来获取一个ModelLoader对象。当时没有再跟进到这个方法的里面再去分析，那么我们现在来看下它的源码：

```
public class Glide {
```

```
public static <T, Y> ModelLoader<T, Y>
buildModelLoader(Class<T> modelClass, Class<Y>
resourceClass,
                 Context context) {
    if (modelClass == null) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Unable to load null model,
setting placeholder only");
        }
        return null;
    }
    return
Glide.get(context).getLoaderFactory().buildModelLoader(mo
delClass, resourceClass);
}

public static Glide get(Context context) {
    if (glide == null) {
        synchronized (Glide.class) {
            if (glide == null) {
                Context applicationContext =
context.getApplicationContext();
                List<GlideModule> modules = new
ManifestParser(applicationContext).parse();
                GlideBuilder builder = new
GlideBuilder(applicationContext);
                for (GlideModule module : modules) {

module.applyOptions(applicationContext, builder);
            }
            glide = builder.createGlide();
            for (GlideModule module : modules) {

module.registerComponents(applicationContext, glide);
            }
        }
    }
}
```

```
        return glide;
    }

    ...
}
```

这里我们还是只看关键，在第11行去构建ModelLoader对象的时候，先调用了一个Glide.get()方法，而这个方法就是关键。我们可以看到，get()方法中实现的是一个单例功能，而创建Glide对象则是在第24行调用GlideBuilder的createGlide()方法来创建的，那么我们跟到这个方法当中：

```
public class GlideBuilder {

    ...

    Glide createGlide() {
        if (sourceService == null) {
            final int cores = Math.max(1,
Runtime.getRuntime().availableProcessors());
            sourceService = new
FifoPriorityThreadPoolExecutor(cores);
        }
        if (diskCacheService == null) {
            diskCacheService = new
FifoPriorityThreadPoolExecutor(1);
        }
        MemorySizeCalculator calculator = new
MemorySizeCalculator(context);
        if (bitmapPool == null) {
            if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.HONEYCOMB) {
                int size =
calculator.getBitmapPoolSize();
                bitmapPool = new LruBitmapPool(size);
            } else {
                bitmapPool = new BitmapPoolAdapter();
            }
        }
    }
}
```

```
    if (memoryCache == null) {
        memoryCache = new
LruResourceCache(calculator.getMemoryCachesize());
    }
    if (diskCacheFactory == null) {
        diskCacheFactory = new
InternalCacheDiskCacheFactory(context);
    }
    if (engine == null) {
        engine = new Engine(memoryCache,
diskCacheFactory, diskCacheService, sourceService);
    }
    if (decodeFormat == null) {
        decodeFormat = DecodeFormat.DEFAULT;
    }
    return new Glide(engine, memoryCache, bitmapPool,
context, decodeFormat);
}
}
```

这里也就是构建Glide对象的地方了。那么观察第22行，你会发现这里new出了一个LruResourceCache，并把它赋值到了memoryCache这个对象上面。你没有猜错，这个就是Glide实现内存缓存所使用的LruCache对象了。不过我这里并不打算展开来讲LruCache算法的具体实现，如果你感兴趣的话可以自己研究一下它的源码。

现在创建好了LruResourceCache对象只能说是把准备工作做好了，接下来我们就一步步研究Glide中的内存缓存到底是如何实现的。

刚才在Engine的load()方法中我们已经看到了生成缓存Key的代码，而内存缓存的代码其实也是在这里实现的，那么我们重新来看一下Engine类load()方法的完整源码：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {
    ...
}
```

```
    public <T, Z, R> LoadStatus load(Key signature, int
width, int height, DataFetcher<T> fetcher,
                                         DataLoadProvider<T, Z> loadProvider,
Transformation<Z> transformation, ResourceTranscoder<Z,
R> transcoder,
                                         Priority priority, boolean isMemoryCacheable,
DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)
{
    Util.assertMainThread();
    long startTime = LogTime.getLogTime();

    final String id = fetcher.getId();
    EngineKey key = keyFactory.buildKey(id,
signature, width, height, loadProvider.getCacheDecoder(),
loadProvider.getSourceDecoder(),
transformation, loadProvider.getEncoder(),
transcoder,
loadProvider.getSourceEncoder());

    EngineResource<?> cached = loadFromCache(key,
isMemoryCacheable);
    if (cached != null) {
        cb.onResourceReady(cached);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Loaded resource from
cache", startTime, key);
        }
        return null;
    }

    EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
    if (active != null) {
        cb.onResourceReady(active);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Loaded resource from
active resources", startTime, key);
        }
    }
}
```

```
        return null;
    }

    EngineJob current = jobs.get(key);
    if (current != null) {
        current.addCallback(cb);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Added to existing
load", startTime, key);
        }
        return new LoadStatus(cb, current);
    }

    EngineJob engineJob = engineJobFactory.build(key,
isMemoryCacheable);
    DecodeJob<T, Z, R> decodeJob = new DecodeJob<T,
Z, R>(key, width, height, fetcher, loadProvider,
transformation,
        transcoder, diskCacheProvider,
diskCacheStrategy, priority);
    EngineRunnable runnable = new
EngineRunnable(engineJob, decodeJob, priority);
    jobs.put(key, engineJob);
    engineJob.addCallback(cb);
    engineJob.start(runnable);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {
        logWithTimeAndKey("Started new load",
startTime, key);
    }
    return new LoadStatus(cb, engineJob);
}

...
}
```

可以看到，这里在第17行调用了loadFromCache()方法来获取缓存图片，如果获取到就直接调用cb.onResourceReady()方法进行回调。如果没有获取到，则会在第26行调用loadFromActiveResources()方法来获取缓存图片，获取到的话也直接进行回调。只有在两个方法都没有获取到缓存的情况下，才会继续向下执行，从而开启线程来加载图片。

也就是说，Glide的图片加载过程中会调用两个方法来获取内存缓存，loadFromCache()和loadFromActiveResources()。这两个方法中一个使用的就是LruCache算法，另一个使用的就是弱引用。我们来看一下它们的源码：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    private final MemoryCache cache;
    private final Map<Key, WeakReference<EngineResource<?>>>> activeResources;
    ...

    private EngineResource<?> loadFromCache(Key key,
        boolean isMemoryCacheable) {
        if (!isMemoryCacheable) {
            return null;
        }
        EngineResource<?> cached =
        getEngineResourceFromCache(key);
        if (cached != null) {
            cached.acquire();
            activeResources.put(key, new
        ResourceWeakReference(key, cached, getReferenceQueue()));
        }
        return cached;
    }

    private EngineResource<?>
    getEngineResourceFromCache(Key key) {
        Resource<?> cached = cache.remove(key);
```

```
    final EngineResource result;
    if (cached == null) {
        result = null;
    } else if (cached instanceof EngineResource) {
        result = (EngineResource) cached;
    } else {
        result = new EngineResource(cached, true
/*isCacheable*/);
    }
    return result;
}

private EngineResource<?> loadFromActiveResources(Key
key, boolean isMemoryCacheable) {
    if (!isMemoryCacheable) {
        return null;
    }
    EngineResource<?> active = null;
    weakReference<EngineResource<?>> activeRef =
activeResources.get(key);
    if (activeRef != null) {
        active = activeRef.get();
        if (active != null) {
            active.acquire();
        } else {
            activeResources.remove(key);
        }
    }
    return active;
}
...
}
```

在loadFromCache()方法的一开始，首先就判断了isMemoryCacheable是不是false，如果是false的话就直接返回null。这是什么意思呢？其实很简单，我们刚刚不是学了一个skipMemoryCache()方法吗？如果在这个方法中传入true，那么这里的isMemoryCacheable就会是false，表示内存缓存已被禁用。

我们继续往下看，接着调用了getEngineResourceFromCache()方法来获取缓存。在这个方法中，会使用缓存Key来从cache当中取值，而这里的cache对象就是在构建Glide对象时创建的LruResourceCache，那么说明这里其实使用的就是LruCache算法了。

但是呢，观察第22行，当我们从LruResourceCache中获取到缓存图片之后会将它从缓存中移除，然后在第16行将这个缓存图片存储到activeResources当中。activeResources就是一个弱引用的HashMap，用来缓存正在使用中的图片，我们可以看到，loadFromActiveResources()方法就是从activeResources这个HashMap当中取值的。使用activeResources来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

好的，从内存缓存中读取数据的逻辑大概就是这些了。概括一下来说，就是如果能从内存缓存当中读取到要加载的图片，那么就直接进行回调，如果读取不到的话，才会开启线程执行后面的图片加载逻辑。

现在我们已经搞明白了内存缓存读取的原理，接下来的问题就是内存缓存是在哪里写入的呢？这里我们又要回顾一下上一篇文章中的内容了。还记得我们之前分析过，当图片加载完成之后，会在EngineJob当中通过Handler发送一条消息将执行逻辑切回到主线程当中，从而执行handleResultOnMainThread()方法。那么我们现在重新来看一下这个方法，代码如下所示：

```
class EngineJob implements  
EngineRunnable.EngineRunnableManager {  
  
    private final EngineResourceFactory  
    engineResourceFactory;  
  
    ...
```

```
private void handleResultOnMainThread() {
    if (isCancelled) {
        resource.recycle();
        return;
    } else if (cbs.isEmpty()) {
        throw new IllegalStateException("Received a
resource without any callbacks to notify");
    }
    engineResource =
engineResourceFactory.build(resource, isCacheable);
    hasResource = true;
    engineResource.acquire();
    listener.onEngineJobComplete(key,
engineResource);
    for (ResourceCallback cb : cbs) {
        if (!isIgnoredCallbacks(cb)) {
            engineResource.acquire();
            cb.onResourceReady(engineResource);
        }
    }
    engineResource.release();
}

static class EngineResourceFactory {
    public <R> EngineResource<R> build(Resource<R>
resource, boolean isMemoryCacheable) {
        return new EngineResource<R>(resource,
isMemoryCacheable);
    }
}
...
}
```

在第13行，这里通过EngineResourceFactory构建出了一个包含图片资源的EngineResource对象，然后会在第16行将这个对象回调到Engine的onEngineJobComplete()方法当中，如下所示：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {
    ...
    @Override
    public void onEngineJobComplete(Key key,
        EngineResource<?> resource) {
        Util.assertMainThread();
        // A null resource indicates that the load
        failed, usually due to an exception.
        if (resource != null) {
            resource.setResourceListener(key, this);
            if (resource.isCacheable()) {
                activeResources.put(key, new
                    ResourceWeakReference(key, resource,
                        getReferenceQueue()));
            }
        }
        jobs.remove(key);
    }
    ...
}
```

现在就非常明显了，可以看到，在第13行，回调过来的EngineResource被put到了activeResources当中，也就是在这里写入的缓存。

那么这只是弱引用缓存，还有另外一种LruCache缓存是在哪里写入的呢？这就要介绍一下EngineResource中的一个引用机制了。观察刚才的handleResultOnMainThread()方法，在第15行和第19行有调用EngineResource的acquire()方法，在第23行有调用它的release()方法。其实，EngineResource是用一个acquired变量用来记录图片被引用的次数，调用acquire()方法会让变量加1，调用release()方法会让变量减1，代码如下所示：

```
class EngineResource<Z> implements Resource<Z> {
```

```
private int acquired;  
...  
  
void acquire() {  
    if (isRecycled) {  
        throw new IllegalStateException("Cannot  
acquire a recycled resource");  
    }  
    if  
(!Looper.getMainLooper().equals(Looper.myLooper())) {  
        throw new IllegalThreadStateException("Must  
call acquire on the main thread");  
    }  
    ++acquired;  
}  
  
void release() {  
    if (acquired <= 0) {  
        throw new IllegalStateException("Cannot  
release a recycled or not yet acquired resource");  
    }  
    if  
(!Looper.getMainLooper().equals(Looper.myLooper())) {  
        throw new IllegalThreadStateException("Must  
call release on the main thread");  
    }  
    if (--acquired == 0) {  
        listener.onResourceReleased(key, this);  
    }  
}
```

也就是说，当acquired变量大于0的时候，说明图片正在使用中，也就应该放到activeResources弱引用缓存当中。而经过release()之后，如果acquired变量等于0了，说明图片已经不再被使用了，那么此时会在第24行调用listener的onResourceReleased()方法来释放资源，这个listener

就是Engine对象，我们来看下它的onResourceReleased()方法：

```
public class Engine implements EngineJobListener,
    MemoryCache.ResourceRemovedListener,
    EngineResource.ResourceListener {

    private final MemoryCache cache;
    private final Map<Key, WeakReference<EngineResource>?>>> activeResources;
    ...

    @Override
    public void onResourceReleased(Key cacheKey,
        EngineResource resource) {
        Util.assertMainThread();
        activeResources.remove(cacheKey);
        if (resource.isCacheable()) {
            cache.put(cacheKey, resource);
        } else {
            resourceRecycler.recycle(resource);
        }
    }
    ...
}
```

可以看到，这里首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

这就是Glide内存缓存的实现原理。

硬盘缓存

接下来我们开始学习硬盘缓存方面的内容。

不知道你还记不记得，在本系列的第一篇文章中我们就使用过硬盘缓存的功能了。当时为了禁止Glide对图片进行硬盘缓存而使用了如下代码：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

调用diskCacheStrategy()方法并传入DiskCacheStrategy.NONE，就可以禁用掉Glide的硬盘缓存功能了。

这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收四种参数：

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.SOURCE： 表示只缓存原始图片。
- DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。
- DiskCacheStrategy.ALL： 表示既缓存原始图片，也缓存转换过后的图片。

上面四种参数的解释本身并没有什么难理解的地方，但是有一个概念大家需要了解，就是当我们使用Glide去加载一张图片的时候，Glide默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在后面学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。而Glide默认情况下在硬盘缓存的就是转换过后的图片，我们通过调用diskCacheStrategy()方法则可以改变这一默认行为。

好的，关于Glide硬盘缓存的用法也就只有这么多，那么接下来还是老套路，我们通过阅读源码来分析一下，Glide的硬盘缓存功能是如何实现的。

首先，和内存缓存类似，硬盘缓存的实现也是使用的LruCache算法，而且Google还提供了一个现成的工具类DiskLruCache。我之前也专门写过一篇文章对这个DiskLruCache工具进行了比较全面的分析，感兴趣的朋友可以参考一下 [Android DiskLruCache完全解析，硬盘缓存的最佳方案](#)。当然，Glide是使用的自己编写的DiskLruCache工具类，但是基本的实现原理都是差不多的。

接下来我们看一下Glide是在哪里读取硬盘缓存的。这里又需要回忆一下上篇文章中的内容了，Glide开启线程来加载图片后会执行EngineRunnable的run()方法，run()方法中又会调用一个decode()方法，那么我们重新再来看一下这个decode()方法的源码：

```
private Resource<?> decode() throws Exception {
    if (isDecodingFromCache()) {
        return decodeFromCache();
    } else {
        return decodeFromSource();
    }
}
```

可以看到，这里会分为两种情况，一种是调用decodeFromCache()方法从硬盘缓存当中读取图片，一种是调用decodeFromSource()来读取原始图片。默认情况下Glide会优先从缓存当中读取，只有缓存中不存在要读取的图片时，才会去读取原始图片。那么我们现在来看一下decodeFromCache()方法的源码，如下所示：

```
private Resource<?> decodeFromCache() throws Exception {
    Resource<?> result = null;
    try {
        result = decodeJob.decodeResultFromCache();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Exception decoding result from
cache: " + e);
        }
    }
    if (result == null) {
        result = decodeJob.decodeSourceFromCache();
    }
    return result;
}
```

可以看到，这里会先去调用DecodeJob的decodeResultFromCache()方法来获取缓存，如果获取不到，会再调用decodeSourceFromCache()方法获取缓存，这两个方法的区别其实也就是DiskCacheStrategy.RESULT和DiskCacheStrategy.SOURCE这两个参数的区别，相信不需要我再做什么解释吧。

那么我们来看一下这两个方法的源码吧，如下所示：

```
public Resource<Z> decodeResultFromCache() throws
Exception {
    if (!diskCacheStrategy.cacheResult()) {
        return null;
    }
    long startTime = LogTime.getLogTime();
    Resource<T> transformed = loadFromCache(resultKey);
    startTime = LogTime.getLogTime();
    Resource<Z> result = transcode(transformed);
    return result;
}

public Resource<Z> decodeSourceFromCache() throws
Exception {
    if (!diskCacheStrategy.cacheSource()) {
        return null;
    }
    long startTime = LogTime.getLogTime();
    Resource<T> decoded =
loadFromCache(resultKey.getOriginalKey());
    return transformEncodeAndTranscode(decoded);
}
```

可以看到，它们都是调用了loadFromCache()方法从缓存当中读取数据，如果是decodeResultFromCache()方法就直接将数据解码并返回，如果是decodeSourceFromCache()方法，还要调用一下transformEncodeAndTranscode()方法先将数据转换一下再解码并返回。

然而我们注意到，这两个方法中在调用loadFromCache()方法时传入的参数却不一样，一个传入的是resultKey，另外一个却又调用了resultKey的getOriginalKey()方法。这个其实非常好理解，刚才我们已经解释过了，Glide的缓存Key是由10个参数共同组成的，包括图片的width、height等等。但如果我们将缓存的原始图片，其实并不需要这么多的参数，因为不用对图片做任何的变化。那么我们来看一下getOriginalKey()方法的源码：

```
public Key getOriginalKey() {  
    if (originalKey == null) {  
        originalKey = new OriginalKey(id, signature);  
    }  
    return originalKey;  
}
```

可以看到，这里其实就是忽略了绝大部分的参数，只使用了id和signature这两个参数来构成缓存Key。而signature参数绝大多数情况下都是用不到的，因此基本上可以说就是由id（也就是图片url）来决定的Original缓存Key。

搞明白了这两种缓存Key的区别，那么接下来我们看一下loadFromCache()方法的源码吧：

```
private Resource<T> loadFromCache(Key key) throws  
IOException {  
    File cacheFile =  
diskCacheProvider.getDiskCache().get(key);  
    if (cacheFile == null) {  
        return null;  
    }  
    Resource<T> result = null;  
    try {  
        result =  
loadProvider.getCacheDecoder().decode(cacheFile, width,  
height);  
    } finally {  
        if (result == null) {
```

```
        diskCacheProvider.getDiskCache().delete(key);
    }
}
return result;
}
```

这个方法的逻辑非常简单，调用getDiskCache()方法获取到的就是Glide自己编写的DiskLruCache工具类的实例，然后调用它的get()方法并把缓存Key传入，就能得到硬盘缓存的文件了。如果文件为空就返回null，如果文件不为空则将它解码成Resource对象后返回即可。

这样我们就将硬盘缓存读取的源码分析完了，那么硬盘缓存又是在哪里写入的呢？趁热打铁我们赶快继续分析下去。

刚才已经分析过了，在没有缓存的情况下，会调用decodeFromSource()方法来读取原始图片。那么我们来看下这个方法：

```
public Resource<Z> decodeFromSource() throws Exception {
    Resource<T> decoded = decodeSource();
    return transformEncodeAndTranscode(decoded);
}
```

这个方法中只有两行代码，decodeSource()顾名思义是用来解析原图片的，而transformEncodeAndTranscode()则是用来对图片进行转换和转码的。我们先来看decodeSource()方法：

```
private Resource<T> decodeSource() throws Exception {
    Resource<T> decoded = null;
    try {
        long startTime = LogTime.getLogTime();
        final A data = fetcher.loadData(priority);
        if (isCancelled) {
            return null;
        }
        decoded = decodeFromSourceData(data);
    } finally {
```

```
        fetcher.cleanup();
    }
    return decoded;
}

private Resource<T> decodeFromSourceData(A data) throws
IOException {
    final Resource<T> decoded;
    if (diskCacheStrategy.cacheSource()) {
        decoded = cacheAndDecodeSourceData(data);
    } else {
        long startTime = LogTime.getLogTime();
        decoded =
loadProvider.getSourceDecoder().decode(data, width,
height);
    }
    return decoded;
}

private Resource<T> cacheAndDecodeSourceData(A data)
throws IOException {
    long startTime = LogTime.getLogTime();
    Sourcewriter<A> writer = new Sourcewriter<A>
(loadProvider.getSourceEncoder(), data);

diskCacheProvider.getDiskCache().put(resultKey.getOriginalKey(),
writer);
    startTime = LogTime.getLogTime();
    Resource<T> result =
loadFromCache(resultKey.getOriginalKey());
    return result;
}
```

这里会在第5行先调用fetcher的loadData()方法读取图片数据，然后在第9行调用decodeFromSourceData()方法来对图片进行解码。接下来会在第18行先判断是否允许缓存原始图片，如果允许的话又会调用cacheAndDecodeSourceData()方法。而在的方法中同样调用了

getDiskCache()方法来获取DiskLruCache实例，接着调用它的put()方法就可以写入硬盘缓存了，注意原始图片的缓存Key是用的resultKey.getOriginalKey()。

好的，原始图片的缓存写入就是这么简单，接下来我们分析一下transformEncodeAndTranscode()方法的源码，来看看转换过后的图片缓存是怎么写入的。代码如下所示：

```
private Resource<Z>
transformEncodeAndTranscode(Resource<T> decoded) {
    long startTime = LogTime.getLogTime();
    Resource<T> transformed = transform(decoded);
    writeTransformedToCache(transformed);
    startTime = LogTime.getLogTime();
    Resource<Z> result = transcode(transformed);
    return result;
}

private void writeTransformedToCache(Resource<T>
transformed) {
    if (transformed == null ||
!diskCacheStrategy.cacheResult()) {
        return;
    }
    long startTime = LogTime.getLogTime();
    SourceWriter<Resource<T>> writer = new
SourceWriter<Resource<T>>(loadProvider.getEncoder(),
transformed);
    diskCacheProvider.getDiskCache().put(resultKey,
writer);
}
```

这里的逻辑就更加简单明了了。先是在第3行调用transform()方法来对图片进行转换，然后在writeTransformedToCache()方法中将转换过后的图片写入到硬盘缓存中，调用的同样是DiskLruCache实例的put()方法，不过这里用的缓存Key是resultKey。

这样我们就将Glide硬盘缓存的实现原理也分析完了。虽然这些源码看上去如此的复杂，但是经过Glide出色的封装，使得我们只需要通过skipMemoryCache()和diskCacheStrategy()这两个方法就可以轻松自如地控制Glide的缓存功能了。

了解了Glide缓存的实现原理之后，接下来我们再来学习一些Glide缓存的高级技巧吧。

高级技巧

虽说Glide将缓存功能高度封装之后，使得用法变得非常简单，但同时也带来了一些问题。

比如之前有一位群里的朋友就跟我说过，他们项目的图片资源都是存放在七牛云上面的，而七牛云为了对图片资源进行保护，会在图片url地址的基础之上再加上一个token参数。也就是说，一张图片的url地址可能会是如下格式：

```
http://url.com/image.jpg?token=d9caa6e02c990b0a
```

而使用Glide加载这张图片的话，也就会使用这个url地址来组成缓存Key。

但是接下来问题就来了，token作为一个验证身份的参数并不是一成不变的，很有可能时时刻刻都在变化。而如果token变了，那么图片的url也就跟着变了，图片url变了，缓存Key也就跟着变了。结果就造成了，明明是同一张图片，就因为token不断在改变，导致Glide的缓存功能完全失效了。

这其实是个挺棘手的问题，而且我相信绝对不仅仅是七牛云这一个个例，大家在使用Glide的时候很有可能都会遇到这个问题。

那么该如何解决这个问题呢？我们还是从源码的层面进行分析，首先再来看一下Glide生成缓存Key这部分的代码：

```
public class Engine implements EngineJobListener,  
    MemoryCache.ResourceRemovedListener,  
    EngineResource.ResourceListener {
```

```
public <T, Z, R> LoadStatus load(Key signature, int width, int height, DataFetcher<T> fetcher, DataLoadProvider<T, Z> loadProvider, Transformation<Z> transformation, ResourceTranscoder<Z, R> transcoder, Priority priority, boolean isMemoryCacheable, DiskCacheStrategy diskCacheStrategy, ResourceCallback cb) {  
    util.assertMainThread();  
    long startTime = LogTime.getLogTime();  
  
    final String id = fetcher.getId();  
    EngineKey key = keyFactory.buildKey(id, signature, width, height, loadProvider.getCacheDecoder(),  
        loadProvider.getSourceDecoder(), transformation, loadProvider.getEncoder(),  
        transcoder,  
    loadProvider.getSourceEncoder());  
  
    ...  
}  
...  
}
```

来看一下第11行，刚才已经说过了，这个id其实就是图片的url地址。那么，这里是通过调用fetcher.getId()方法来获取的图片url地址，而我们在上一篇文章中已经知道了，fetcher就是HttpUrlFetcher的实例，我们就来看一下它的getId()方法的源码吧，如下所示：

```
public class HttpUrlFetcher implements  
DataFetcher<InputStream> {  
  
    private final GlideUrl glideUrl;  
    ...  
  
    public HttpUrlFetcher(GlideUrl glideUrl) {  
        this(glideUrl, DEFAULT_CONNECTION_FACTORY);
```

```
}

HttpUrlFetcher(GlideUrl glideUrl,
HttpURLConnectionFactory connectionFactory) {
    this.glideUrl = glideUrl;
    this.connectionFactory = connectionFactory;
}

@Override
public String getId() {
    return glideUrl.getCacheKey();
}

...
}
```

可以看到，`getId()`方法中又调用了`GlideUrl`的`getCacheKey()`方法。那么这个`GlideUrl`对象是从哪里来的呢？其实就是在`load()`方法中传入的图片url地址，然后Glide在内部把这个url地址包装成了一个`GlideUrl`对象。

很明显，接下来我们就要看一下`GlideUrl`的`getCacheKey()`方法的源码了，如下所示：

```
public class GlideUrl {

    private final URL url;
    private final String urlString;
    ...

    public GlideUrl(URL url) {
        this(url, Headers.DEFAULT);
    }

    public GlideUrl(String url) {
        this(url, Headers.DEFAULT);
    }
}
```

```
public GlideUrl(URL url, Headers headers) {  
    ...  
    this.url = url;  
    stringUrl = null;  
}  
  
public GlideUrl(String url, Headers headers) {  
    ...  
    this.stringUrl = url;  
    this.url = null;  
}  
  
public String getCacheKey() {  
    return stringUrl != null ? stringUrl :  
url.toString();  
}  
...  
}
```

这里我将代码稍微进行了一点简化，这样看上去更加简单明了。GlideUrl类的构造函数接收两种类型的参数，一种是url字符串，一种是URL对象。然后getCacheKey()方法中的判断逻辑非常简单，如果传入的是url字符串，那么就直接返回这个字符串本身，如果传入的是URL对象，那么就返回这个对象toString()后的结果。

其实看到这里，我相信大家已经猜到解决方案了，因为getCacheKey()方法中的逻辑太直白了，直接就是将图片的url地址进行返回来作为缓存Key的。那么其实我们只需要重写这个getCacheKey()方法，加入一些自己的逻辑判断，就能轻松解决掉刚才的问题了。

创建一个MyGlideUrl继承自GlideUrl，代码如下所示：

```
public class MyGlideUrl extends GlideUrl {  
    private String mUrl;
```

```
public MyGlideUrl(String url) {
    super(url);
    mUrl = url;
}

@Override
public String getCacheKey() {
    return mUrl.replace(findTokenParam(), "");
}

private String findTokenParam() {
    String tokenParam = "";
    int tokenKeyIndex = mUrl.indexOf("?token=") >= 0
? mUrl.indexOf("?token=") : mUrl.indexOf("&token=");
    if (tokenKeyIndex != -1) {
        int nextAndIndex = mUrl.indexOf("&",
tokenKeyIndex + 1);
        if (nextAndIndex != -1) {
            tokenParam = mUrl.substring(tokenKeyIndex
+ 1, nextAndIndex + 1);
        } else {
            tokenParam =
mUrl.substring(tokenKeyIndex);
        }
    }
    return tokenParam;
}

}
```

可以看到，这里我们重写了getCacheKey()方法，在里面加入了一段逻辑用于将图片url地址中token参数的这一部分移除掉。这样getCacheKey()方法得到的就是一个没有token参数的url地址，从而不管token怎么变化，最终Glide的缓存Key都是固定不变的了。

当然，定义好了MyGlideUrl，我们还得使用它才行，将加载图片的代码改成如下方式即可：

```
Glide.with(this)
    .load(new MyGlideUrl(url))
    .into(imageview);
```

也就是说，我们需要在load()方法中传入这个自定义的MyGlideUrl对象，而不能再像之前那样直接传入url字符串了。不然的话Glide在内部还是会使用原始的GlideUrl类，而不是我们自定义的MyGlideUrl类。

这样我们就将这个棘手的缓存问题给解决掉了。

好了，关于Glide缓存方面的内容今天就分析到这里，现在我们不光掌握了Glide缓存的基本用法和高级技巧，还了解了它背后的实现原理，又是收获满满的一篇文章啊。下一篇文章当中，我会继续带着大家深入分析Glide的其他功能模块，讲一讲回调方面的知识，感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（四），玩转Glide的回调与监听](#)。

3.13Android图片加载框架最全解析（四），玩转Glide的回调与监听

大家好，今天我们继续学习Glide。

在上一篇文章当中，我带着大家一起深入探究了Glide的缓存机制，我们不光掌握了Glide缓存的使用方法，还通过源码分析对缓存的工作原理进行了了解。虽说上篇文章和本篇文章的内容关系并不是很大，不过感兴趣的朋友还是可以去阅读一下 [Android图片加载框架最全解析（三），深入探究Glide的缓存机制](#)。

今天是这个Glide系列的第四篇文章，我们又要选取一个新的功能模块开始学习了，那么就来研究一下Glide的回调和监听功能吧。今天的学习模式仍然是以基本用法和源码分析相结合的方式来进行的，当然，本文中的源码还是建在第二篇源码分析的基础之上，还没有看过这篇文章的朋友，建议先去阅读 [Android图片加载框架最全解析（二），从源码的角度理解Glide的执行流程](#)。

回调的源码实现

作为一名Glide老手，相信大家对于Glide的基本用法已经非常熟练了。我们都清楚，使用Glide在界面上加载并展示一张图片只需要一行代码：

```
Glide.with(this).load(url).into(imageview);
```

而在这一行代码的背后，Glide帮我们执行了成千上万行的逻辑。其实在第二篇文章当中，我们已经分析了这一行代码背后的完整执行流程，但是这里我准备再带着大家单独回顾一下回调这部分的源码，这将有助于我们今天这篇文章的学习。

首先来看一下into()方法，这里我们将ImageView的实例传入到into()方法当中，Glide将图片加载完成之后，图片就能显示到ImageView上了。这是怎么实现的呢？我们来看一下into()方法的源码：

```
public Target<TranscodeType> into(ImageView view) {  
    Util.assertMainThread();  
    if (view == null) {  
        throw new IllegalArgumentException("You must pass  
in a non null view");  
    }  
    if (!isTransformationSet && view.getScaleType() !=  
null) {  
        switch (view.getScaleType()) {  
            case CENTER_CROP:  
                applyCenterCrop();  
                break;  
            case FIT_CENTER:  
            case FIT_START:  
            case FIT_END:  
                applyFitCenter();  
                break;  
            default:  
                // Do nothing.  
        }  
    }  
    return into(glide.buildImageViewTarget(view,  
transcodeClass));  
}
```

可以看到，最后一行代码会调用glide.buildImageViewTarget()方法构建出一个Target对象，然后再把它传入到另一个接收Target参数的into()方法中。Target对象则是用来最终展示图片用的，如果我们跟进到glide.buildImageViewTarget()方法中，你会看到如下的源码：

```
public class ImageViewTargetFactory {

    @SuppressWarnings("unchecked")
    public <Z> Target<Z> buildTarget(ImageView view,
Class<Z> clazz) {
        if (GlideDrawable.class.isAssignableFrom(clazz))
{
            return (Target<Z>) new
GlideDrawableImageViewTarget(view);
        } else if (Bitmap.class.equals(clazz)) {
            return (Target<Z>) new
BitmapImageViewTarget(view);
        } else if
(Drawable.class.isAssignableFrom(clazz)) {
            return (Target<Z>) new
DrawableImageViewTarget(view);
        } else {
            throw new IllegalArgumentException("Unhandled
class: " + clazz
                    + ", try .as*
(Class).transcode(ResourceTranscoder)");
        }
    }
}
```

buildTarget()方法会根据传入的class参数来构建不同的Target对象，如果你在使用Glide加载图片的时候调用了asBitmap()方法，那么这里就会构建出BitmapImageViewTarget对象，否则的话构建的都是GlideDrawableImageViewTarget对象。至于上述代码中的DrawableImageViewTarget对象，这个通常都是用不到的，我们可以暂时不用管它。

之后就会把这里构建出来的Target对象传入到GenericRequest当中，而Glide在图片加载完成之后又会回调GenericRequest的onResourceReady()方法，我们来看一下这部分源码：

```
public final class GenericRequest<A, T, Z, R> implements Request, SizeReadyCallback, ResourceCallback {

    private Target<R> target;
    ...

    private void onResourceReady(Resource<?> resource, R result) {
        boolean isFirstResource = isFirstReadyResource();
        status = Status.COMPLETE;
        this.resource = resource;
        if (requestListener == null ||
            !requestListener.onResourceReady(result, model, target,
                loadedFromMemoryCache, isFirstResource))
        {
            GlideAnimation<R> animation =
            animationFactory.build(loadedFromMemoryCache,
            isFirstResource);
            target.onResourceReady(result, animation);
        }
        notifyLoadSuccess();
    }
    ...
}
```

这里在第14行调用了target.onResourceReady()方法，而刚才我们已经知道，这里的target就是GlideDrawableImageViewTarget对象，那么我们再来看一下它的源码：

```
public class GlideDrawableImageViewTarget extends ImageViewTarget<GlideDrawable> {
    ...
}
```

```
@Override
public void onResourceReady(GlideDrawable resource,
GlideAnimation<? super GlideDrawable> animation) {
    if (!resource.isAnimated()) {
        float viewRatio = view.getWidth() / (float)
view.getHeight();
        float drawableRatio =
resource.getIntrinsicWidth() / (float)
resource.getIntrinsicHeight();
        if (Math.abs(viewRatio - 1f) <=
SQUARE_RATIO_MARGIN
            && Math.abs(drawableRatio - 1f) <=
SQUARE_RATIO_MARGIN) {
            resource = new SquaringDrawable(resource,
view.getWidth());
        }
    }
    super.onResourceReady(resource, animation);
    this.resource = resource;
    resource.setLoopCount(maxLoopCount);
    resource.start();
}

@Override
protected void setResource(GlideDrawable resource) {
    view.setImageDrawable(resource);
}

...
}
```

可以看到，这里在onResourceReady()方法中处理了图片展示，还有GIF播放的逻辑，那么一张图片也就显示出来了，这也就是Glide回调的基本实现原理。

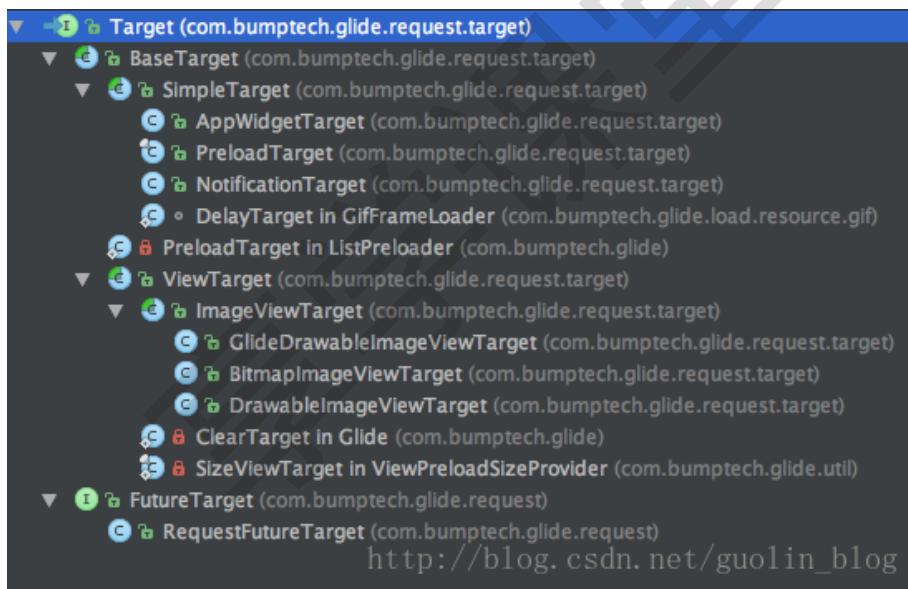
好的，那么原理就先分析到这儿，接下来我们就来看一下在回调和监听方面还有哪些知识是可以扩展的。

into()方法

使用了这么久的Glide，我们都应该知道into()方法中是可以传入ImageView的。那么into()方法还可以传入别的参数吗？我可以让Glide加载出来的图片不显示到ImageView上吗？答案是肯定的，这就需要用到自定义Target功能。

其实通过上面的分析，我们已经知道了，into()方法还有一个接收Target参数的重载。即使我们传入的参数是ImageView，Glide也会在内部自动构建一个Target对象。而如果我们能够掌握自定义Target技术的话，就可以更加随心所欲地控制Glide的回调了。

我们先来看一下Glide中Target的继承结构图吧，如下所示：



可以看到，Target的继承结构还是相当复杂的，实现Target接口的子类非常多。不过你不用被这么多的子类所吓到，这些大多数都是Glide已经实现好的具备完整功能的Target子类，如果我们要进行自定义的话，通常只需要在两种Target的基础上去自定义就可以了，一种是SimpleTarget，一种是ViewTarget。

接下来我就分别以这两种Target来举例，学习一下自定义Target的功能。

首先来看SimpleTarget，顾名思义，它是一种极为简单的Target，我们使用它可以将Glide加载出来的图片对象获取到，而不是像之前那样只能将图片在ImageView上显示出来。

那么下面我们来看一下SimpleTarget的用法示例吧，其实非常简单：

```
simpleTarget<GlideDrawable> simpleTarget = new
SimpleTarget<GlideDrawable>() {
    @Override
    public void onResourceReady(GlideDrawable resource,
        GlideAnimation glideAnimation) {
        imageView.setImageDrawable(resource);
    }
};

public void loadImage(View view) {
    String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
    Glide.with(this)
        .load(url)
        .into(simpleTarget);
}
```

怎么样？不愧是SimpleTarget吧，短短几行代码就搞了。这里我们创建了一个SimpleTarget的实例，并且指定它的泛型是GlideDrawable，然后重写了onResourceReady()方法。在onResourceReady()方法中，我们就可获取到Glide加载出来的图片对象了，也就是方法参数中传过来的GlideDrawable对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了ImageView上。

SimpleTarget的实现创建好了，那么只需要在加载图片的时候将它传入到into()方法中就可以了，现在运行一下程序，效果如下图所示。



虽然目前这个效果和直接在into()方法中传入ImageView并没有什么区别，但是我们已经拿到了图片对象的实例，然后就可以随意做更多的事情了。

当然，SimpleTarget中的泛型并不一定只能是GlideDrawable，如果你能确定你正在加载的是一张静态图而不是GIF图的话，我们还能直接拿到这张图的Bitmap对象，如下所示：

```
SimpleTarget<Bitmap> simpleTarget = new
SimpleTarget<Bitmap>() {
    @Override
    public void onResourceReady(Bitmap resource,
GlideAnimation glideAnimation) {
        imageView.setImageBitmap(resource);
    }
};

public void loadImage(View view) {
    String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
    Glide.with(this)
        .load(url)
```

```
.asBitmap()
.into(simpleTarget);
}
```

可以看到，这里我们将SimpleTarget的泛型指定成Bitmap，然后在加载图片的时候调用了asBitmap()方法强制指定这是一张静态图，这样就能在onResourceReady()方法中获取到这张图的Bitmap对象了。

好了，SimpleTarget的用法就是这么简单，接下来我们学习一下ViewTarget的用法。

事实上，从刚才的继承结构图上就能看出，Glide在内部自动帮我们创建的GlideDrawableImageViewTarget就是ViewTarget的子类。只不过GlideDrawableImageViewTarget被限定只能作用在ImageView上，而ViewTarget的功能更加广泛，它可以作用在任意的View上。

这里我们还是通过一个例子来演示一下吧，比如我创建了一个自定义布局MyLayout，如下所示：

```
public class MyLayout extends LinearLayout {

    private ViewTarget<MyLayout, GlideDrawable>
viewTarget;

    public MyLayout(Context context, AttributeSet attrs)
{
    super(context, attrs);
    viewTarget = new ViewTarget<MyLayout,
GlideDrawable>(this) {
        @Override
        public void onResourceReady(GlideDrawable
resource, GlideAnimation glideAnimation) {
            MyLayout myLayout = getView();
            myLayout.setImageAsBackground(resource);
        }
    };
}
```

```
public ViewTarget<MyLayout, GlideDrawable>
getTarget() {
    return viewTarget;
}

public void setImageAsBackground(GlideDrawable
resource) {
    setBackground(resource);
}

}
```

在MyLayout的构造函数中，我们创建了一个ViewTarget的实例，并将MyLayout当前的实例this传了进去。ViewTarget中需要指定两个泛型，一个是View的类型，一个图片的类型（GlideDrawable或Bitmap）。然后在onResourceReady()方法中，我们就可以通过getView()方法获取到MyLayout的实例，并调用它的任意接口了。比如说这里我们调用了setImageAsBackground()方法来将加载出来的图片作为MyLayout布局的背景图。

接下来看一下怎么使用这个Target吧，由于MyLayout中已经提供了getTarget()接口，我们只需要在加载图片的地方这样写就可以了：

```
public class MainActivity extends AppCompatActivity {

    MyLayout myLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myLayout = (MyLayout)
        findViewById(R.id.background);
    }

    public void loadImage(View view) {
```

```
        String url =
"http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-
CN7336795473_1920x1080.jpg";
        Glide.with(this)
            .load(url)
            .into(myLayout.getTarget());
    }

}
```

就是这么简单，在into()方法中传入myLayout.getTarget()即可。现在重新运行一下程序，效果如下图所示。



好的，关于自定义Target的功能我们就介绍这么多，这些虽说都是自定义Target最基本的用法，但掌握了这些用法之后，你就能应对各种各样复杂的逻辑了。

preload()方法

Glide加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待漫长的网络加载时间了，这该怎么办呢？

对于很多Glide新手来说这确实是一个烦恼的问题，因为在没有学习本篇文章之前，into()方法中必须传入一个ImageView呀，而传了ImageView之后图片就显示出来了，这还怎么预加载呢？

不过在学习了本篇文章之后，相信你已经能够想到解决方案了。因为into()方法中除了传入ImageView之后还可以传入Target对象，如果我们在Target对象的onResourceReady()方法中做一个空实现，也就是不做任何逻辑处理，那么图片自然也就显示不出来了，而Glide的缓存机制却仍然还会正常工作，这样不就实现预加载功能了吗？

没错，上述的做法完全可以实现预加载功能，不过有没有感觉这种实现方式有点笨笨的。事实上，Glide专门给我们提供了预加载的接口，也就是preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换into()方法即可，如下所示：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .preload();
```

需要注意的是，我们如果使用了preload()方法，最好要将diskCacheStrategy的缓存策略指定成DiskCacheStrategy.SOURCE。因为preload()方法默认是预加载的原始图片大小，而into()方法则默认会根据ImageView控件的大小来动态决定加载图片的大小。因此，如果不将diskCacheStrategy的缓存策略指定成DiskCacheStrategy.SOURCE的话，很容易会造成我们在预加载完成之后再使用into()方法加载图片，却仍然还是要从网络上去请求图片这种现象。

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为Glide会直接从缓存当中去读取图片并显示出来，代码如下所示：

```
Glide.with(this)
    .load(url)
    .diskCacheStrategy(DiskCacheStrategy.SOURCE)
    .into(imageview);
```

注意，这里我们仍然需要使用diskCacheStrategy()方法将硬盘缓存策略指定成DiskCacheStrategy.SOURCE，以保证Glide一定会去读取刚才预加载的图片缓存。

preload()方法的用法大概就是这么简单，但是仅仅会使用显然层次有些太低了，下面我们就满足一下好奇心，看看它的源码是如何实现的。

和into()方法一样，preload()方法也是在GenericRequestBuilder类当中的，代码如下所示：

```
public class GenericRequestBuilder<ModelType, DataType,
ResourceType, TranscodeType> implements Cloneable {
    ...
    public Target<TranscodeType> preload(int width, int
height) {
        final PreloadTarget<TranscodeType> target =
PreloadTarget.obtain(width, height);
        return into(target);
    }

    public Target<TranscodeType> preload() {
        return preload(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL);
    }

    ...
}
```

正如刚才所说，preload()方法有两个方法重载，你可以调用带参数的preload()方法来明确指定图片的宽和高，也可以调用不带参数的preload()方法，它会在内部自动将图片的宽和高都指定成Target.SIZE_ORIGINAL，也就是图片的原始尺寸。

然后我们可以看到，这里在第5行调用了PreloadTarget.obtain()方法获取一个PreloadTarget的实例，并把它传入到了into()方法当中。从刚才的继承结构图中可以看出，PreloadTarget是SimpleTarget的子类，因此它是可以直接传入到into()方法中的。

那么现在的问题就是，PreloadTarget具体的实现到底是什么样子的了，我们看一下它的源码，如下所示：

```
public final class PreloadTarget<Z> extends  
SimpleTarget<Z> {  
  
    public static <Z> PreloadTarget<Z> obtain(int width,  
    int height) {  
        return new PreloadTarget<Z>(width, height);  
    }  
  
    private PreloadTarget(int width, int height) {  
        super(width, height);  
    }  
  
    @Override  
    public void onResourceReady(Z resource,  
    GlideAnimation<? super Z> glideAnimation) {  
        Glide.clear(this);  
    }  
}
```

PreloadTarget的源码非常简单，obtain()方法中就是new了一个PreloadTarget的实例而已，而onResourceReady()方法中也没做什么事情，只是调用了Glide.clear()方法。

这里的Glide.clear()并不是清空缓存的意思，而是表示加载已完成，释放资源的意思，因此不用在这里产生疑惑。

其实PreloadTarget的思想和我们刚才提到设计思路是一样的，就是什么都不做就可以了。因为图片加载完成之后只将它缓存而不去显示它，那不就相当于预加载了嘛。

preload()方法不管是在用法方面还是源码实现方面都还是非常简单的，那么关于这个方法我们就学到这里。

downloadOnly()方法

一直以来，我们使用Glide都是为了将图片显示到界面上。虽然我们知道Glide会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实Glide将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到downloadOnly()方法了。

和preload()方法类似，downloadOnly()方法也是可以替换into()方法的，不过downloadOnly()方法的用法明显要比preload()方法复杂不少。顾名思义，downloadOnly()方法表示只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。downloadOnly()方法是定义在DrawableTypeRequest类当中的，它有两个方法重载，一个接收图片的宽度和高度，另一个接收一个泛型对象，如下所示：

- downloadOnly(int width, int height)
- downloadOnly(Y target)

这两个方法各自有各自的应用场景，其中downloadOnly(int width, int height)是用于在子线程中下载图片的，而downloadOnly(Y target)是用于在主线程中下载图片的。

那么我们先来看downloadOnly(int width, int height)的用法。当调用了downloadOnly(int width, int height)方法后会立即返回一个FutureTarget对象，然后Glide会在后台开始下载图片文件。接下来我们调用FutureTarget的get()方法就可以去获取下载好的图片文件了，如果此时图片还没有下载完，那么get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

```
public void downloadImage(View view) {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                String url =  
                    "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
                    CN7336795473_1920x1080.jpg";  
                final Context context =  
                    getApplicationContext();  
                FutureTarget<File> target =  
                    Glide.with(context)  
  
                    .load(url)  
  
                    .downloadOnly(Target.SIZE_ORIGINAL,  
                    Target.SIZE_ORIGINAL);  
                final File imageFile = target.get();  
                runOnUiThread(new Runnable() {  
                    @Override  
                    public void run() {  
                        Toast.makeText(context,  
                            imageFile.getPath(), Toast.LENGTH_LONG).show();  
                    }  
                });  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
}).start();  
}
```

这段代码稍微有一点点长，我带着大家解读一下。首先刚才说了，`downloadOnly(int width, int height)`方法必须要用在子线程当中，因此这里的一步就是new了一个Thread。在子线程当中，我们先获取了一个Application Context，这个时候不能再用Activity作为Context了，因为会有Activity销毁了但子线程还没执行完这种可能出现。

接下来就是Glide的基本用法，只不过将`into()`方法替换成`downloadOnly()`方法。`downloadOnly()`方法会返回一个`FutureTarget`对象，这个时候其实Glide已经开始在后台下载图片了，我们随时都可以调用`FutureTarget`的`get()`方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的`File`对象返回。

最后，我们使用`runOnUiThread()`切回到主线程，然后使用`Toast`将下载好的图片文件路径显示出来。

现在重新运行一下代码，效果如下图所示。



这样我们就能清晰地看出来图片完整的缓存路径是什么了。

之后我们可以使用如下代码去加载这张图片，图片就会立即显示出来，而不用再去网络上请求了：

```
public void loadImage(View view) {  
    String url =  
    "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
    CN7336795473_1920x1080.jpg";  
    Glide.with(this)  
        .load(url)  
        .diskCacheStrategy(DiskCacheStrategy.SOURCE)  
        .into(imageView);  
}
```

需要注意的是，这里必须将硬盘缓存策略指定成 DiskCacheStrategy.SOURCE 或者 DiskCacheStrategy.ALL，否则 Glide 将无法使用我们刚才下载好的图片缓存文件。

现在重新运行一下代码，效果如下图所示。



可以看到，图片的加载和显示是非常快的，因为 Glide 直接使用的是刚才下载好的缓存文件。

那么这个 downloadOnly(int width, int height) 方法的工作原理到底是什么样的呢？我们来简单快速地看一下它的源码吧。

首先在DrawableTypeRequest类当中可以找到定义这个地方，如下所示：

```
public class DrawableTypeRequest<ModelType> extends  
DrawableRequestBuilder<ModelType>  
implements DownloadOptions {  
    ...  
  
    public FutureTarget<File> downloadOnly(int width, int  
height) {  
        return  
getDownloadOnlyRequest().downloadOnly(width, height);  
    }  
  
    private GenericTranscodeRequest<ModelType,  
InputStream, File> getDownloadOnlyRequest() {  
        return optionsApplier.apply(new  
GenericTranscodeRequest<ModelType, InputStream, File>(  
        File.class, this, streamModelLoader,  
InputStream.class, File.class, optionsApplier));  
    }  
}
```

这里会先调用getDownloadOnlyRequest()方法得到一个GenericTranscodeRequest对象，然后再调用它的downloadOnly()方法，代码如下所示：

```
public class GenericTranscodeRequest<ModelType, DataType,  
ResourceType>  
implements DownloadOptions {  
    ...  
  
    public FutureTarget<File> downloadOnly(int width, int  
height) {  
        return getDownloadOnlyRequest().into(width,  
height);  
    }  
}
```

```
    private GenericRequestBuilder<ModelType, DataType, File, File> getDownloadOnlyRequest() {
        ResourceTranscoder<File, File> transcoder =
UnitTranscoder.get();
        DataLoadProvider<DataType, File> dataLoadProvider
= glide.buildDataProvider(dataClass, File.class);
        FixedLoadProvider<ModelType, DataType, File,
File> fixedLoadProvider =
            new FixedLoadProvider<ModelType, DataType,
File, File>(modelLoader, transcoder, dataLoadProvider);
        return optionsApplier.apply(
            new GenericRequestBuilder<ModelType,
DataType, File, File>(fixedLoadProvider,
File.class, this))
.priority(Priority.LOW)

.diskCacheStrategy(DiskCacheStrategy.SOURCE)
.skipMemoryCache(true);
    }
}
```

这里又是调用了一个getDownloadOnlyRequest()方法来构建了一个图片下载的请求，getDownloadOnlyRequest()方法会返回一个GenericRequestBuilder对象，接着调用它的into(width, height)方法，我们继续跟进去瞧一瞧：

```
public FutureTarget<TranscodeType> into(int width, int
height) {
    final RequestFutureTarget<ModelType, TranscodeType>
target =
        new RequestFutureTarget<ModelType,
TranscodeType>(glide.getMainHandler(), width, height);
    glide.getMainHandler().post(new Runnable() {
        @Override
        public void run() {
            if (!target.isCancelled()) {
                into(target);

```

```
        }
    }
});

return target;
}
```

可以看到，这里首先是new出了一个RequestFutureTarget对象，RequestFutureTarget也是Target的子类之一。然后通过Handler将线程切回到主线程当中，再将这个RequestFutureTarget传入到into()方法当中。

那么也就是说，其实这里就是调用了接收Target参数的into()方法，然后Glide就开始执行正常的图片加载逻辑了。那么现在剩下的问题就是，这个RequestFutureTarget中到底处理了些什么逻辑？我们打开它的源码看一看：

```
public class RequestFutureTarget<T, R> implements
FutureTarget<R>, Runnable {
    ...

    @Override
    public R get() throws InterruptedException,
ExecutionException {
    try {
        return doGet(null);
    } catch (TimeoutException e) {
        throw new AssertionError(e);
    }
}

    @Override
    public R get(long time, TimeUnit timeUnit) throws
InterruptedException, ExecutionException,
TimeoutException {
    return doGet(timeUnit.toMillis(time));
}
```

```
    @Override
    public void getSize(SizeReadyCallback cb) {
        cb.onSizeReady(width, height);
    }

    @Override
    public synchronized void onLoadFailed(Exception e,
    Drawable errorDrawable) {
        exceptionReceived = true;
        this.exception = e;
        waiter.notifyAll(this);
    }

    @Override
    public synchronized void onResourceReady(R resource,
    GlideAnimation<? super R> glideAnimation) {
        resultReceived = true;
        this.resource = resource;
        waiter.notifyAll(this);
    }

    private synchronized R doGet(Long timeoutMillis)
throws ExecutionException, InterruptedException,
    TimeoutException {
        if (assertBackgroundThread) {
            util.assertBackgroundThread();
        }

        if (isCancelled) {
            throw new CancellationException();
        } else if (exceptionReceived) {
            throw new ExecutionException(exception);
        } else if (resultReceived) {
            return resource;
        }

        if (timeoutMillis == null) {
            waiter.waitForTimeout(this, 0);
        } else {
            waiter.waitForTimeout(this, timeoutMillis);
        }
    }
}
```

```
        } else if (timeoutMillis > 0) {
            waiter.waitForTimeout(this, timeoutMillis);
        }

        if (Thread.interrupted()) {
            throw new InterruptedException();
        } else if (exceptionReceived) {
            throw new ExecutionException(exception);
        } else if (isCancelled) {
            throw new CancellationException();
        } else if (!resultReceived) {
            throw new TimeoutException();
        }

        return resource;
    }

    static class Waiter {

        public void waitForTimeout(Object toWaitOn, long
timeoutMillis) throws InterruptedException {
            toWaitOn.wait(timeoutMillis);
        }

        public void notifyAll(Object toNotify) {
            toNotify.notifyAll();
        }
    }

    ...
}
```

这里我对RequestFutureTarget的源码做了一些精简，我们只看最主要的逻辑就可以了。

刚才我们已经学习过了downloadOnly()方法的基本用法，在调用了downloadOnly()方法之后，再调用FutureTarget的get()方法，就能获取到下载的图片文件了。而downloadOnly()方法返回的FutureTarget对象其实就是这个RequestFutureTarget，因此我们直接来看它的get()方法就行了。

RequestFutureTarget的get()方法中又调用了一个doGet()方法，而doGet()方法才是真正处理具体逻辑的地方。首先在doGet()方法中会判断当前是否是在子线程当中，如果不是的话会直接抛出一个异常。然后下面会判断下载是否已取消、或者已失败，如果是已取消或者已失败的话都会直接抛出一个异常。接下来会根据resultReceived这个变量来判断下载是否已完成，如果这个变量为true的话，就直接把结果进行返回。

那么如果下载还没有完成呢？我们继续往下看，接下来就进入到一个wait()当中，把当前线程给阻塞住，从而阻止代码继续往下执行。这也是为什么downloadOnly(int width, int height)方法要求必须在子线程当中使用，因为它会对当前线程进行阻塞，如果在主线程当中使用的话，那么就会让主线程卡死，从而用户无法进行任何其他操作。

那么现在线程被阻塞住了，什么时候才能恢复呢？答案在onResourceReady()方法中。可以看到，onResourceReady()方法中只有三行代码，第一行把resultReceived赋值成true，说明图片文件已经下载好了，这样下次再调用get()方法时就不会再阻塞线程，而是可以直接将结果返回。第二行把下载好的图片文件赋值到一个全局的resource变量上面，这样doGet()方法就可以访问到它。第三行notifyAll一下，通知所有wait的线程取消阻塞，这个时候图片文件已经下载好了，因此doGet()方法也就可以返回结果了。

好的，这就是downloadOnly(int width, int height)方法的基本用法和实现原理，那么下面我们来看一下downloadOnly(Y target)方法。

回想一下，其实downloadOnly(int width, int height)方法必须使用在子线程当中，最主要还是因为它在内部帮我们自动创建了一个RequestFutureTarget，是这个RequestFutureTarget要求必须在子线程当中执行。而downloadOnly(Y target)方法则要求我们传入一个自己创建的Target，因此就不受RequestFutureTarget的限制了。

但是downloadOnly(Y target)方法的用法也会相对更复杂一些，因为我们又要自己创建一个Target了，而且这次必须直接去实现最顶层的Target接口，比之前的SimpleTarget和ViewTarget都要复杂不少。

那么下面我们就来实现一个最简单的DownloadImageTarget吧，注意Target接口的泛型必须指定成File对象，这是downloadOnly(Y target)方法要求的，代码如下所示：

```
public class DownloadImageTarget implements Target<File>
{
    private static final String TAG =
    "DownloadImageTarget";

    @Override
    public void onStart() {
    }

    @Override
    public void onStop() {
    }

    @Override
    public void onDestroy() {
    }

    @Override
    public void onLoadStarted(Drawable placeholder) {
    }

    @Override
    public void onLoadFailed(Exception e, Drawable
errorDrawable) {
    }

    @Override
    public void onResourceReady(File resource,
GlideAnimation<? super File> glideAnimation) {
```

```
        Log.d(TAG, resource.getPath());
    }

    @Override
    public void onLoadCleared(Drawable placeholder) {
    }

    @Override
    public void getSize(SizeReadyCallback cb) {
        cb.onSizeReady(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL);
    }

    @Override
    public void setRequest(Request request) {
    }

    @Override
    public Request getRequest() {
        return null;
    }
}
```

由于是要直接实现Target接口，因此需要重写的方法非常多。这些方法大多是Glide加载图片生命周期的一些回调，我们可以不用管它们，其中只有两个方法是必须实现的，一个是getSize()方法，一个是onResourceReady()方法。

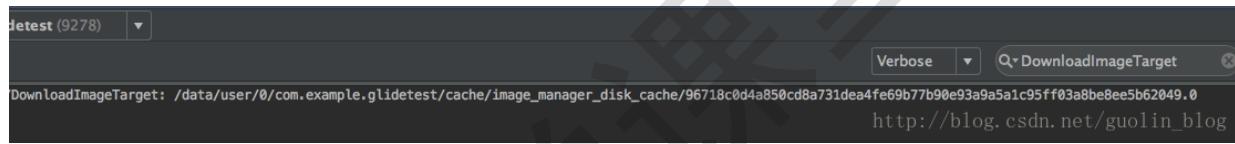
在第二篇Glide源码解析的时候，我带着大家一起分析过，Glide在开始加载图片之前会先计算图片的大小，然后回调到onSizeReady()方法当中，之后才会开始执行图片加载。而这里，计算图片大小的任务就交给我们了。只不过这是一个最简单的Target实现，我在getSize()方法中就直接回调了Target.SIZE_ORIGINAL，表示图片的原始尺寸。

然后onResourceReady()方法我们就非常熟悉了，图片下载完成之后就会回调到这里，我在这个方法中只是打印了一下下载的图片文件的路径。

这样一个最简单的DownloadImageTarget就定义好了，使用它也非常的简单，我们不用再考虑什么线程的问题了，而是直接把它的实例传入downloadOnly(Y target)方法中即可，如下所示：

```
public void downloadImage(View view) {  
    String url =  
        "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
        CN7336795473_1920x1080.jpg";  
    Glide.with(this)  
        .load(url)  
        .downloadOnly(new DownloadImageTarget());  
}
```

现在重新运行一下代码并点击Download Image按钮，然后观察控制台日志的输出，结果如下图所示。



这样我们就使用了downloadOnly(Y target)方法同样获取到下载的图片文件的缓存路径了。

好的，那么关于downloadOnly()方法我们就学到这里。

listener()方法

今天学习的内容已经够多了，下面我们就以一个简单的知识点结尾吧，Glide回调与监听的最后一部分——listener()方法。

其实listener()方法的作用非常普遍，它可以用来监听Glide加载图片的状态。举个例子，比如说我们刚才使用了preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果Glide加载图片失败了，我该怎样调试错误的原因呢？答案都在listener()方法当中。

首先来看下listener()方法的基本用法吧，不同于刚才几个方法都是要替换into()方法的，listener()是结合into()方法一起使用的，当然也可以结合preload()方法一起使用。最基本的用法如下所示：

```
public void loadImage(View view) {  
    String url =  
    "http://cn.bing.com/az/hprichbg/rb/TOAD_ZH-  
    CN7336795473_1920x1080.jpg";  
    Glide.with(this)  
        .load(url)  
        .listener(new RequestListener<String,  
    GlideDrawable>() {  
            @Override  
            public boolean onException(Exception e,  
                String model, Target<GlideDrawable> target,  
                boolean isFirstResource) {  
                return false;  
            }  
  
            @Override  
            public boolean  
            onResourceReady(GlideDrawable resource, String model,  
                Target<GlideDrawable> target, boolean  
                isFromMemoryCache, boolean isFirstResource) {  
                return false;  
            }  
        })  
        .into(imageView);  
}
```

这里我们在into()方法之前串接了一个listener()方法，然后实现了一个RequestListener的实例。其中RequestListener需要实现两个方法，一个onResourceReady()方法，一个onException()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调onResourceReady()方法，而当图片加载失败的时候就会回调onException()方法，onException()方法中会将失败的Exception参数传进来，这样我们就可以定位具体失败的原因了。

没错，`listener()`方法就是这么简单。不过还有一点需要处理，`onResourceReady()`方法和`onException()`方法都有一个布尔值的返回值，返回`false`就表示这个事件没有被处理，还会继续向下传递，返回`true`就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单的例子，如果我们在`RequestListener`的`onResourceReady()`方法中返回了`true`，那么就不会再回调`Target`的`onResourceReady()`方法了。

关于`listener()`方法的用法就讲这么多，不过还是老规矩，我们再来看一下它的源码是怎么实现的吧。

首先，`listener()`方法是定义在`GenericRequestBuilder`类当中的，而我们传入到`listener()`方法中的实例则会赋值到一个`requestListener`变量当中，如下所示：

```
public class GenericRequestBuilder<ModelType, DataType,
    ResourceType, TranscodeType> implements Cloneable {

    private RequestListener<? super ModelType,
        TranscodeType> requestListener;

    ...

    public GenericRequestBuilder<ModelType, DataType,
        ResourceType, TranscodeType> listener(
            RequestListener<? super ModelType,
        TranscodeType> requestListener) {
        this.requestListener = requestListener;
        return this;
    }

    ...

}
```

接下来在构建`GenericRequest`的时候这个变量也会被一起传进去，最后在图片加载完成的时候，我们会看到如下逻辑：

```
public final class GenericRequest<A, T, Z, R> implements
    Request, SizeReadyCallback,
    ResourceCallback {
```

```
    private RequestListener<? super A, R>
requestListener;

    ...

    private void onResourceReady(Resource<?> resource, R
result) {
    boolean isFirstResource = isFirstReadyResource();
    status = Status.COMPLETE;
    this.resource = resource;
    if (requestListener == null ||
!requestListener.onResourceReady(result, model, target,
        loadedFromMemoryCache, isFirstResource))
{
    GlideAnimation<R> animation =
animationFactory.build(loadedFromMemoryCache,
isFirstResource);
    target.onResourceReady(result, animation);
}
    notifyLoadSuccess();
}
...
}
```

可以看到，这里在第11行会先回调requestListener的onResourceReady()方法，只有当这个onResourceReady()方法返回false的时候，才会继续调用Target的onResourceReady()方法，这也就是listener()方法的实现原理。

另外一个onException()方法的实现机制也是一模一样的，代码同样是在GenericRequest类中，如下所示：

```
public final class GenericRequest<A, T, Z, R> implements
Request, SizeReadyCallback,
ResourceCallback {
    ...
}
```

```
    @Override
    public void onException(Exception e) {
        status = Status.FAILED;
        if (requestListener == null ||
            !requestListener.onException(e, model,
target, isFirstReadyResource())) {
            setErrorPlaceholder(e);
        }
    }

    ...
}
```

可以看到，这里会在第9行回调requestListener的onException()方法，只有在onException()方法返回false的情况下才会继续调用setErrorPlaceholder()方法。也就是说，如果我们在onException()方法中返回了true，那么Glide请求中使用error(int resourceId)方法设置的异常占位图就失效了。

这样我们也就将listener()方法的全部实现原理都分析完了。

好了，关于Glide回调与监听方面的内容今天就讲到这里，这一篇文章的内容非常充实，希望大家都能好好掌握。下一篇文章当中，我会继续带着大家深入分析Glide的其他功能模块，讲一讲图片变换方面的知识，感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（五），Glide强大的图片变换功能](#)。

3.14Android图片加载框架最全解析（六），探究Glide的自定义模块功能

不知不觉中，我们的Glide系列教程已经到了第六篇了，距离第一篇Glide的基本用法发布已经过去了半年的时间。在这半年中，我们通过用法讲解和源码分析配合学习的方式，将Glide的方方面面都研究了个遍，相信一直能看到这里的朋友现在已经是一位Glide高手了。

整个Glide系列预计总共会有八篇文章，现在也是逐步进入尾声了。不过，越是到后面，我们探究的内容也越是更加深入。那么今天，我们就来一起探究一下Glide中一个比较深入，但同时也是非常重要的一个功能——自定义模块。

自定义模块的基本用法

学到这里相信你已经知道，Glide的用法是非常非常简单的，大多数情况下，我们想要实现的图片加载效果只需要一行代码就能解决了。但是Glide过于简洁的API也造成了一个问题，就是如果我们想要更改Glide的某些默认配置项应该怎么操作呢？很难想象如何将更改Glide配置项的操作串联到一行经典的Glide图片加载语句中当中吧？没错，这个时候就需要用到自定义模块功能了。

自定义模块功能可以将更改Glide配置，替换Glide组件等操作独立出来，使得我们能轻松地对Glide的各种配置进行自定义，并且又和Glide的图片加载逻辑没有任何交集，这也是一种低耦合编程方式的体现。那么接下来我们就学习一下自定义模块的基本用法。

首先需要定义一个我们自己的模块类，并让它实现GlideModule接口，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
    glide) {
    }
}
```

可以看到，在MyGlideModule类当中，我们重写了applyOptions()和registerComponents()方法，这两个方法分别就是用来更改Glide和配置以及替换Glide组件的。我们待会儿只需要在这两个方法中加入具体的逻辑，就能实现更改Glide配置或者替换Glide组件的功能了。

不过，目前Glide还无法识别我们自定义的MyGlideModule，如果想要让它生效，还得在AndroidManifest.xml文件当中加入如下配置才行：

```
<manifest>
    ...
<application>
    <meta-data
        android:name="com.example.glidetest.MyGlideModule"
        android:value="GlideModule" />
    ...
</application>
</manifest>
```

在标签中加入一个meta-data配置项，其中android:name指定成我们自定义的MyGlideModule的完整路径， android:value必须指定成 GlideModule，这个是固定值。

这样的话，我们就将Glide自定义模块的功能完成了，是不是非常简单？现在Glide已经能够识别我们自定义的这个MyGlideModule了，但是在编写具体的功能之前，我们还是按照老规矩阅读一下源码，从源码的层面上来分析一下， Glide到底是如何识别出这个自定义的MyGlideModule的。

自定义模块的原理

这里我不会带着大家从Glide代码执行的第一步一行行重头去解析Glide的源码，而是只分析和自定义模块相关的部分。如果你想将Glide的源码通读一遍的话，可以去看本系列的第二篇文章 [Android图片加载框架最全解析（二），从源码的角度理解Glide的执行流程](#)。

显然我们已经用惯了

`Glide.with(context).load(url).into(imageView)`这样一行简洁的Glide图片加载语句，但是我们好像从来没有注意过Glide这个类本身实例。然而事实上，Glide类确实是有创建实例的，只不过是在内部由Glide自动帮我们创建和管理了，对于开发者而言，大多数情况下是不用关心它的，只需要调用它的静态方法就可以了。

那么Glide的实例到底是在哪里创建的呢？我们来看下Glide类中的`get()`方法的源码，如下所示：

```
public class Glide {  
  
    private static volatile Glide glide;  
  
    ...  
  
    public static Glide get(Context context) {  
        if (glide == null) {  
            synchronized (Glide.class) {  
                if (glide == null) {  
                    Context applicationContext =  
context.getApplicationContext();  
                    List<GlideModule> modules = new  
ManifestParser(applicationContext).parse();  
                    GlideBuilder builder = new  
GlideBuilder(applicationContext);  
                    for (GlideModule module : modules) {  
  
module.applyOptions(applicationContext, builder);  
                }  
                glide = builder.createGlide();  
                for (GlideModule module : modules) {  
            }  
        }  
    }  
}
```

```
        module.registerComponents(applicationContext, glide);
    }
}
}
return glide;
}

...
}
```

我们来仔细看一下上面这段代码。首先这里使用了一个单例模式来获取Glide对象的实例，可以看到，这是一个非常典型的双重锁模式。然后在第12行，调用ManifestParser的parse()方法去解析AndroidManifest.xml文件中的配置，实际上就是将AndroidManifest中所有值为GlideModule的meta-data配置读取出来，并将相应的自定义模块实例化。由于你可以自定义任意多个模块，因此这里我们将会得到一个GlideModule的List集合。

接下来在第13行创建了一个GlideBuilder对象，并通过一个循环调用了每一个GlideModule的applyOptions()方法，同时也把GlideBuilder对象作为参数传入到这个方法中。而applyOptions()方法就是我们可以加入自己的逻辑的地方了，虽然目前为止我们还没有编写任何逻辑。

再往下的一步就非常关键了，这里调用了GlideBuilder的createGlide()方法，并返回了一个Glide对象。也就是说，Glide对象的实例就是在这里创建的了，那么我们跟到这个方法当中瞧一瞧：

```
public class GlideBuilder {
    private final Context context;

    private Engine engine;
    private BitmapPool bitmapPool;
    private MemoryCache memoryCache;
    private ExecutorService sourceService;
    private ExecutorService diskCacheService;
    private DecodeFormat decodeFormat;
```

```
private DiskCache.Factory diskCacheFactory;

...

Glide createGlide() {
    if (sourceService == null) {
        final int cores = Math.max(1,
Runtime.getRuntime().availableProcessors());
        sourceService = new
FifoPriorityThreadPoolExecutor(cores);
    }
    if (diskCacheService == null) {
        diskCacheService = new
FifoPriorityThreadPoolExecutor(1);
    }
    MemorySizeCalculator calculator = new
MemorySizeCalculator(context);
    if (bitmapPool == null) {
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.HONEYCOMB) {
            int size =
calculator.getBitmapPoolSize();
            bitmapPool = new LruBitmapPool(size);
        } else {
            bitmapPool = new BitmapPoolAdapter();
        }
    }
    if (memoryCache == null) {
        memoryCache = new
LruResourceCache(calculator.getMemoryCachesize());
    }
    if (diskCacheFactory == null) {
        diskCacheFactory = new
InternalCacheDiskCacheFactory(context);
    }
    if (engine == null) {
        engine = new Engine(memoryCache,
diskCacheFactory, diskCacheService, sourceService);
    }
}
```

```
    }

    if (decodeFormat == null) {
        decodeFormat = DecodeFormat.DEFAULT;
    }

    return new Glide(engine, memoryCache, bitmapPool,
context, decodeFormat);
}

}
```

这个方法中会创建BitmapPool、MemoryCache、DiskCache、DecodeFormat等对象的实例，并在最后一行创建一个Glide对象的实例，然后将前面创建的这些实例传入到Glide对象当中，以供后续的图片加载操作使用。

但是大家有没有注意到一个细节，createGlide()方法中创建任何对象的时候都做了一个空检查，只有在对象为空的时候才会去创建它的实例。也就是说，如果我们在applyOptions()方法中提前就给这些对象初始化并赋值，那么在createGlide()方法中就不会再去重新创建它们的实例了，从而也就实现了更改Glide配置的功能。关于这个功能我们待会儿会进行具体的演示。

现在继续回到Glide的get()方法中，得到了Glide对象的实例之后，接下来又通过一个循环调用了每一个GlideModule的registerComponents()方法，在这里我们可以加入替换Glide的组件的逻辑。

好了，这就是Glide自定义模块的全部工作原理。了解了它的工作原理之后，接下来所有的问题就集中在我们到底如何在applyOptions()和registerComponents()这两个方法中加入具体的逻辑了，下面我们马上就要学习一下。

更改Glide配置

刚才在分析自定义模式工作原理的时候其实就已经提到了，如果想要更改Glide的默认配置，其实只需要在applyOptions()方法中提前将Glide的配置项进行初始化就可以了。那么Glide一共有哪些配置项呢？这里我给大家做了一个列举：

- `setMemoryCache()`
用于配置Glide的内存缓存策略， 默认配置是LruResourceCache。
- `setBitmapPool()`
用于配置Glide的Bitmap缓存池， 默认配置是LruBitmapPool。
- `setDiskCache()`
用于配置Glide的硬盘缓存策略， 默认配置是
`InternalCacheDiskCacheFactory`。
- `setDiskCacheService()`
用于配置Glide读取缓存中图片的异步执行器， 默认配置是
`FifoPriorityThreadPoolExecutor`， 也就是先入先出原则。
- `setResizeService()`
用于配置Glide读取非缓存中图片的异步执行器， 默认配置也是
`FifoPriorityThreadPoolExecutor`。
- `setDecodeFormat()`
用于配置Glide加载图片的解码模式， 默认配置是RGB_565。

其实Glide的这些默认配置都非常科学且合理，使用的缓存算法也都是效率极高的，因此在绝大多数情况下我们并不需要去修改这些默认配置，这也是Glide用法能如此简洁的一个原因。

但是Glide科学的默认配置并不影响我们去学习自定义Glide模块的功能，因此总有某些情况下，默认的配置可能将无法满足你，这个时候就需要我们自己动手来修改默认配置了。

下面就通过具体的实例来看一下吧。刚才说到， Glide默认的硬盘缓存策略使用的是`InternalCacheDiskCacheFactory`，这种缓存会将所有Glide加载的图片都存储到当前应用的私有目录下。这是一种非常安全的做法，但同时这种做法也造成了一些不便，因为私有目录下即使是开发者自己也是无法查看的，如果我想要去验证一下图片到底有没有成功缓存下来，这就有点不太好办了。

这种情况下，就非常适合使用自定义模块来更改Glide的默认配置。我们可以自己去实现`DiskCache.Factory`接口来自定义一个硬盘缓存策略，不过却大大没有必要这么做，因为Glide本身就内置了一个`ExternalCacheDiskCacheFactory`，可以允许将加载的图片都缓存到SD卡。

那么接下来，我们就尝试使用这个ExternalCacheDiskCacheFactory来替换默认的InternalCacheDiskCacheFactory，从而将所有Glide加载的图片都缓存到SD卡上。

由于在前面我们已经创建好了一个自定义模块MyGlideModule，那么现在就可以直接在这里编写逻辑了，代码如下所示：

```
public class MyGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        builder.setDiskCache(new
ExternalCacheDiskCacheFactory(context));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {

}

}
```

没错，就是这么简单，现在所有Glide加载的图片都会缓存到SD卡上了。

另外，InternalCacheDiskCacheFactory和ExternalCacheDiskCacheFactory的默认硬盘缓存大小都是250M。也就是说，如果你的应用缓存的图片总大小超出了250M，那么Glide就会按照DiskLruCache算法的原则来清理缓存的图片。

当然，我们是可以对这个默认的缓存大小进行修改的，而且修改方式非常简单，如下所示：

```
public class MyGlideModule implements GlideModule {

    public static final int DISK_CACHE_SIZE = 500 * 1024
* 1024;
```

```
    @Override
    public void applyOptions(Context context,
GlideBuilder builder) {
        builder.setDiskCache(new
ExternalCacheDiskCacheFactory(context, DISK_CACHE_SIZE));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {

}

}
```

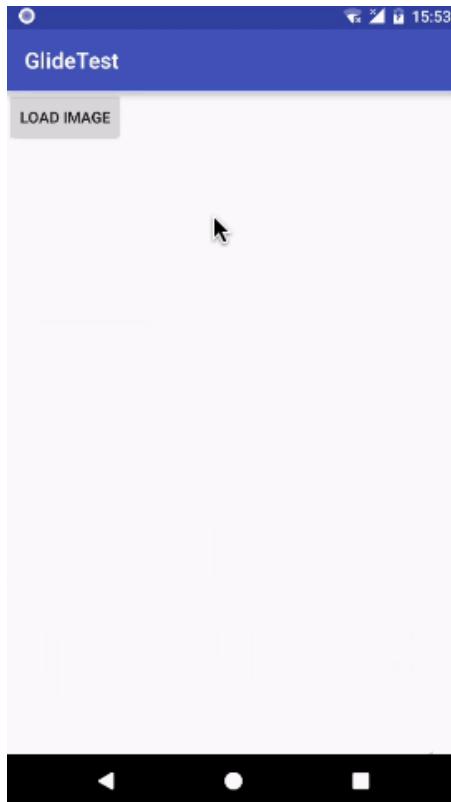
只需要向ExternalCacheDiskCacheFactory或者InternalCacheDiskCacheFactory再传入一个参数就可以了，现在我们就将Glide硬盘缓存的大小调整成了500M。

好了，更改Glide配置的功能就是这么简单，那么接下来我们就来验证一下更改的配置到底有没有生效吧。

这里还是使用最基本的Glide加载语句来去加载一张网络图片：

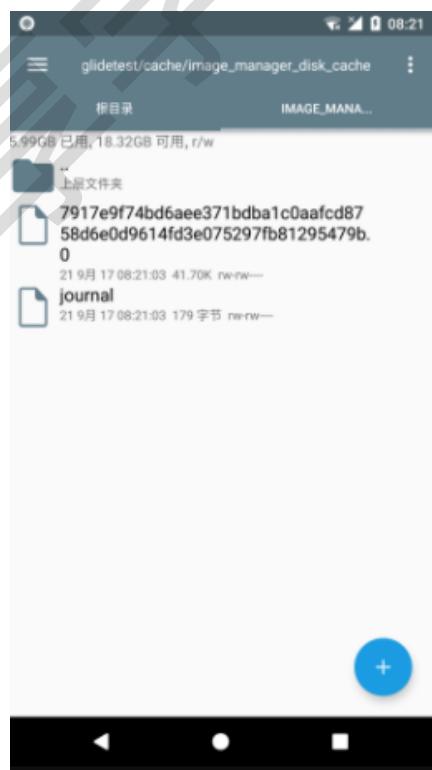
```
String url = "http://guolin.tech/book.png";
Glide.with(this)
    .load(url)
    .into(imageview);
```

运行一下程序，效果如下图所示：



OK，现在图片已经加载出现了，那么我们去找一找它的缓存吧。

ExternalCacheDiskCacheFactory的默认缓存路径是在 `sdcard/Android/包名/cache/image_manager_disk_cache` 目录当中，我们使用文件浏览器进入到这个目录，结果如下图所示。



可以看到，这里有两个文件，其中journal文件是DiskLruCache算法的日志文件，这个文件必不可少，且只会有一个。想了解更多关于DiskLruCache算法的朋友，可以去阅读我的这篇博客 [Android DiskLruCache完全解析，硬盘缓存的最佳方案](#)。

而另外一个文件就是那张缓存的图片了，它的文件名虽然看上去很奇怪，但是我们只需要把这个文件的后缀改成.png，然后用图片浏览器打开，结果就一目了然了，如下图所示。



由此证明，我们已经成功将Glide的硬盘缓存路径修改到SD卡上了。

另外这里再提一点，我们都知道Glide和Picasso的用法是非常相似的，但是有一点差别却很大。Glide加载图片的默认格式是RGB_565，而Picasso加载图片的默认格式是ARGB_8888。ARGB_8888格式的图片效果会更加细腻，但是内存开销会比较大。而RGB_565格式的图片则更加节省内存，但是图片效果上会差一些。

Glide和Picasso各自采取的默认图片格式谈不上孰优孰劣，只能说各自的取舍不一样。但是如果你希望Glide也能使用ARGB_8888的图片格式，这当然也是可以的。我们只需要在MyGlideModule中更改一下默认配置即可，如下所示：

```
public class MyGlideModule implements GlideModule {
```

```
public static final int DISK_CACHE_SIZE = 500 * 1024
* 1024;

@Override
public void applyOptions(Context context,
GlideBuilder builder) {
    builder.setDiskCache(new
ExternalCacheDiskCacheFactory(context, DISK_CACHE_SIZE));

builder.setDecodeFormat(DecodeFormat.PREFER_ARGB_8888);
}

@Override
public void registerComponents(Context context, Glide
glide) {

}

}
```

通过这样配置之后，使用Glide加载的所有图片都将会使用ARGB_8888的格式，虽然图片质量变好了，但同时内存开销也会明显增大，所以你要做好心理准备哦。

好了，关于更改Glide配置的内容就介绍这么多，接下来就让我们进入到下一个非常重要的主题，替换Glide组件。

替换Glide组件

替换Glide组件功能需要在自定义模块的registerComponents()方法中加入具体的替换逻辑。相比于更改Glide配置，替换Glide组件这个功能的难度就明显大了不少。Glide中的组件非常繁多，也非常复杂，但其实大多数情况下并不需要我们去做什么替换。不过，有一个组件却有着比较大的替换需求，那就是Glide的HTTP通讯组件。

默认情况下，Glide使用的是基于原生HttpURLConnection进行订制的HTTP通讯组件，但是现在大多数的Android开发者都更喜欢使用OkHttp，因此将Glide中的HTTP通讯组件修改成OkHttp的这个需求比较常见，那么今天我们也会以这个功能来作为例子进行讲解。

首先来看一下Glide中目前有哪些组件吧，在Glide类的构造方法当中，如下所示：

```
public class Glide {  
  
    Glide(Engine engine, MemoryCache memoryCache,  
    BitmapPool bitmapPool, Context context, DecodeFormat  
    decodeFormat) {  
        ...  
  
        register(File.class, ParcelFileDescriptor.class,  
        new FileDescriptorFileLoader.Factory());  
        register(File.class, InputStream.class, new  
        StreamFileLoader.Factory());  
        register(int.class, ParcelFileDescriptor.class,  
        new FileDescriptorResourceLoader.Factory());  
        register(int.class, InputStream.class, new  
        StreamResourceLoader.Factory());  
        register(Integer.class,  
        ParcelFileDescriptor.class, new  
        FileDescriptorResourceLoader.Factory());  
        register(Integer.class, InputStream.class, new  
        StreamResourceLoader.Factory());  
        register(String.class,  
        ParcelFileDescriptor.class, new  
        FileDescriptorStringLoader.Factory());  
        register(String.class, InputStream.class, new  
        StreamStringLoader.Factory());  
        register(Uri.class, ParcelFileDescriptor.class,  
        new FileDescriptorUriLoader.Factory());  
        register(Uri.class, InputStream.class, new  
        StreamUriLoader.Factory());
```

```
        register(URL.class, InputStream.class, new
StreamUrlLoader.Factory());
        register(GlideUrl.class, InputStream.class, new
HttpUrlGlideUrlLoader.Factory());
        register(byte[].class, InputStream.class, new
StreamByteArrayLoader.Factory());

        ...
}

}
```

可以看到，这里都是以调用register()方法的方式来注册一个组件，register()方法中传入的参数表示Glide支持使用哪种参数类型来加载图片，以及如何去处理这种类型的图片加载。举个例子：

```
register(GlideUrl.class, InputStream.class, new
HttpUrlGlideUrlLoader.Factory());
```

这句代码就表示，我们可以使用`Glide.with(context).load(new GlideUrl("url...")).into(imageView)`的方式来加载图片，而`HttpUrlGlideUrlLoader.Factory`则是要负责处理具体的网络通讯逻辑。如果我们想要将Glide的HTTP通讯组件替换成OkHttp的话，那么只需要在自定义模块当中重新注册一个GlideUrl类型的组件就行了。

说到这里有的朋友可能会疑问了，我们平时使用Glide加载图片时，大多数情况下都是直接将图片的URL字符串传入到load()方法当中的，很少会将它封装成GlideUrl对象之后再传入到load()方法当中，那为什么只需要重新注册一个GlideUrl类型的组件，而不需要去重新注册一个String类型的组件呢？其实道理很简单，因为load(String)方法只是Glide给我们提供一种简易的API封装而已，它的底层仍然还是调用的GlideUrl组件，因此我们在替换组件的时候只需要直接替换最底层的，这样就一步到位了。

那么接下来我们就开始学习到底如何将Glide的HTTP通讯组件替换成OkHttp。

首先第一步，不用多说，肯定是要先将OkHttp的库引入到当前项目中，如下所示：

```
dependencies {  
    compile 'com.squareup.okhttp3:okhttp:3.9.0'  
}
```

然后接下来该怎么做呢？我们只要依葫芦画瓢就可以了。刚才不是说Glide的网络通讯逻辑是由HttpUrlGlideUrlLoader.Factory来负责的吗，那么我们就来看一下它的源码：

```
public class HttpUrlGlideUrlLoader implements  
ModelLoader<.GlideUrl, InputStream> {  
  
    private final ModelCache<.GlideUrl, GlideUrl>  
modelCache;  
  
    public static class Factory implements  
ModelLoaderFactory<.GlideUrl, InputStream> {  
        private final ModelCache<.GlideUrl, GlideUrl>  
modelCache = new ModelCache<.GlideUrl, GlideUrl>(500);  
  
        @Override  
        public ModelLoader<.GlideUrl, InputStream>  
build(Context context, GenericLoaderFactory factories) {  
            return new HttpUrlGlideUrlLoader(modelCache);  
        }  
  
        @Override  
        public void teardown() {  
        }  
    }  
  
    public HttpUrlGlideUrlLoader() {  
        this(null);  
    }  
}
```

```
public HttpUrlGlideUrlLoader(ModelCache<GlideUrl,  
GlideUrl> modelCache) {  
    this.modelCache = modelCache;  
}  
  
@Override  
public DataFetcher<InputStream>  
getResourceFetcher(GlideUrl model, int width, int height)  
{  
    GlideUrl url = model;  
    if (modelCache != null) {  
        url = modelCache.get(model, 0, 0);  
        if (url == null) {  
            modelCache.put(model, 0, 0, model);  
            url = model;  
        }  
    }  
    return new HttpUrlFetcher(url);  
}  
}
```

可以看到，`HttpUrlGlideUrlLoader.Factory`是一个内部类，外层的`HttpUrlGlideUrlLoader`类实现了`ModelLoader<GlideUrl, InputStream>`这个接口，并重写了`getResourceFetcher()`方法。而在`getResourceFetcher()`方法中，又创建了一个`HttpUrlFetcher`的实例，在这里才是真正处理具体网络通讯逻辑的地方，代码如下所示：

```
public class HttpUrlFetcher implements  
DataFetcher<InputStream> {  
    private static final String TAG = "HttpUrlFetcher";  
    private static final int MAXIMUM_REDIRECTS = 5;  
    private static final HttpUrlConnectionFactory  
DEFAULT_CONNECTION_FACTORY = new  
DefaultHttpUrlConnectionFactory();  
  
    private final GlideUrl glideUrl;  
    private final HttpUrlConnectionFactory  
connectionFactory;
```

```
private HttpURLConnection urlConnection;
private InputStream stream;
private volatile boolean isCancelled;

public HttpUrlFetcher(GlideUrl glideUrl) {
    this(glideUrl, DEFAULT_CONNECTION_FACTORY);
}

HttpUrlFetcher(GlideUrl glideUrl,
HttpURLConnectionFactory connectionFactory) {
    this.glideUrl = glideUrl;
    this.connectionFactory = connectionFactory;
}

@Override
public InputStream loadData(Priority priority) throws
Exception {
    return loadDataWithRedirects(glideUrl.toURL(), 0
, null , glideUrl.getHeaders());
}

private InputStream loadDataWithRedirects(URL url,
int redirects, URL lastUrl, Map<String, String> headers)
throws IOException {
    if (redirects >= MAXIMUM_REDIRECTS) {
        throw new IOException("Too many (> " +
MAXIMUM_REDIRECTS + ") redirects!");
    } else {
        try {
            if (lastUrl != null &&
url.toURI().equals(lastUrl.toURI())) {
                throw new IOException("In re-direct
loop");
            }
        } catch (URISyntaxException e) {
        }
    }
}
```

```
urlConnection = connectionFactory.build(url);
for (Map.Entry<String, String> headerEntry :
headers.entrySet()) {

urlConnection.addRequestProperty(headerEntry.getKey(),
headerEntry.getValue());
}

urlConnection.setConnectTimeout(2500);
urlConnection.setReadTimeout(2500);
urlConnection.setUseCaches(false);
urlConnection.connect();
if (isCancelled) {
    return null;
}
final int statusCode =
urlConnection.getResponseCode();
if (statusCode / 100 == 2) {
    return
getStreamForSuccessfulRequest(urlConnection);
} else if (statusCode / 100 == 3) {
    String redirect urlString =
urlConnection.getHeaderField("Location");
    if (TextUtils.isEmpty(redirect urlString)) {
        throw new IOException("Received empty or
null redirect url");
    }
    URL redirectUrl = new URL(url,
redirect urlString);
    return loadDataWithRedirects(redirectUrl,
redirects + 1, url, headers);
} else {
    if (statusCode == -1) {
        throw new IOException("Unable to retrieve
response code from HttpURLConnection.");
    }
    throw new IOException("Request failed " +
statusCode + ":" + urlConnection.getResponseMessage());
}
```

```
}

private InputStream
getStreamForSuccessfulRequest(HttpURLConnection
urlConnection)
        throws IOException {
    if
(TextUtils.isEmpty(urlConnection.getContentEncoding())) {
        int contentLength =
urlConnection.getContentLength();
        stream =
ContentLengthInputStream.obtain(urlConnection.getInputStream(),
contentLength);
    } else {
        stream = urlConnection.getInputStream();
    }
    return stream;
}

@Override
public void cleanup() {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException e) {
        }
    }
    if (urlConnection != null) {
        urlConnection.disconnect();
    }
}

@Override
public String getId() {
    return glideUrl.getCacheKey();
}

@Override
```

```
public void cancel() {
    isCancelled = true;
}

interface HttpURLConnectionFactory {
    HttpURLConnection build(URL url) throws
IOException;
}

private static class DefaultHttpURLConnectionFactory
implements HttpURLConnectionFactory {
    @Override
    public HttpURLConnection build(URL url) throws
IOException {
        return (HttpURLConnection)
url.openConnection();
    }
}
}
```

上面这段代码看上去应该不费力吧？其实就是一些HttpURLConnection的用法而已。那么我们只需要仿照着HttpUrlFetcher的代码来写，并且把HTTP的通讯组件替换成OkHttp就可以了。

现在新建一个OkHttpFetcher类，并且同样实现DataFetcher接口，代码如下所示：

```
public class OkHttpFetcher implements
DataFetcher<InputStream> {

    private final OkHttpClient client;
    private final GlideUrl url;
    private InputStream stream;
    private ResponseBody responseBody;
    private volatile boolean isCancelled;

    public OkHttpFetcher(OkHttpClient client, GlideUrl
url) {
```

```
        this.client = client;
        this.url = url;
    }

    @Override
    public InputStream loadData(Priority priority) throws
Exception {
    Request.Builder requestBuilder = new
Request.Builder()
        .url(url.toStringUrl());
    for (Map.Entry<String, String> headerEntry :
url.getHeaders().entrySet()) {
        String key = headerEntry.getKey();
        requestBuilder.addHeader(key,
headerEntry.getValue());
    }
    requestBuilder.addHeader("httpplib", "okHttp");
    Request request = requestBuilder.build();
    if (isCancelled) {
        return null;
    }
    Response response =
client.newCall(request).execute();
    responseBody = response.body();
    if (!response.isSuccessful() || responseBody ==
null) {
        throw new IOException("Request failed with
code: " + response.code());
    }
    stream =
ContentLengthInputStream.obtain(responseBody.byteStream()
,
responseBody.contentLength());
    return stream;
}

@Override
public void cleanup() {
```

```
try {
    if (stream != null) {
        stream.close();
    }
    if (responseBody != null) {
        responseBody.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}

@Override
public String getId() {
    return url.getCacheKey();
}

@Override
public void cancel() {
    isCancelled = true;
}
}
```

上面这段代码完全就是我照着HttpUrlFetcher依葫芦画瓢写出来的，用的也都是一些OkHttp的基本用法，相信不需要再做什么解释了吧。可以看到，使用OkHttp来编写网络通讯的代码要比使用HttpURLConnection简单很多，代码行数也少了很多。注意在第22行，我添加了一个http://OkHttp的请求头，这个是待会儿我们用来进行测试验证的，大家实际项目中的代码无须添加这个请求头。

那么我们就继续发挥依葫芦画瓢的精神，仿照着HttpUrlGlideUrlLoader再写一个OkHttpGlideUrlLoader吧。新建一个OkHttpGlideUrlLoader类，并且实现ModelLoader<.GlideUrl, InputStream>接口，代码如下所示：

```
public class okHttpGlideUrlLoader implements
ModelLoader<.GlideUrl, InputStream> {
```

```
private OkHttpClient okHttpClient;

public static class Factory implements
ModelLoaderFactory<.GlideUrl, InputStream> {

    private OkHttpClient client;

    public Factory() {
    }

    public Factory(OkHttpClient client) {
        this.client = client;
    }

    private synchronized OkHttpClient
getOkHttpClient() {
        if (client == null) {
            client = new OkHttpClient();
        }
        return client;
    }

    @Override
    public ModelLoader<.GlideUrl, InputStream>
build(Context context, GenericLoaderFactory factories) {
        return new
OkHttp.GlideUrlLoader(getOkHttpClient());
    }

    @Override
    public void teardown() {
    }
}

public OkHttp.GlideUrlLoader(OkHttpClient client) {
    this.okHttpClient = client;
}
```

```
    @Override
    public DataFetcher<InputStream>
getResourceFetcher(GlideUrl model, int width, int height)
{
    return new OkHttpFetcher(okHttpClient, model);
}
}
```

注意这里的Factory我提供了两个构造方法，一个是不带任何参数的，一个是有OkHttpClient参数的。如果对OkHttp不需要进行任何自定义的配置，那么就调用无参的Factory构造函数即可，这样会在内部自动创建一个OkHttpClient实例。但如果你需要添加拦截器，或者修改OkHttp的默认超时等等配置，那么就自己创建一个OkHttpClient的实例，然后传入到Factory的构造方法当中就行了。

好了，现在就只差最后一步，将我们刚刚创建的OkHttpGlideUrlLoader和OkHttpFetcher注册到Glide当中，将原来的HTTP通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {
    ...
    @Override
    public void registerComponents(Context context, Glide
glide) {
        glide.register(GlideUrl.class, InputStream.class,
new OkHttpGlideUrlLoader.Factory());
    }
}
```

可以看到，这里也是调用了Glide的register()方法来注册组件的。register()方法中使用的Map类型来存储已注册的组件，因此我们这里重新注册了一遍GlideUrl.class类型的组件，就把原来的组件给替换掉了。

理论上来说，现在我们已经成功将Glide的HTTP通讯组件替换成OkHttp了，现在唯一的问题就是我们该如何去验证一下到底有没有替换成功呢？

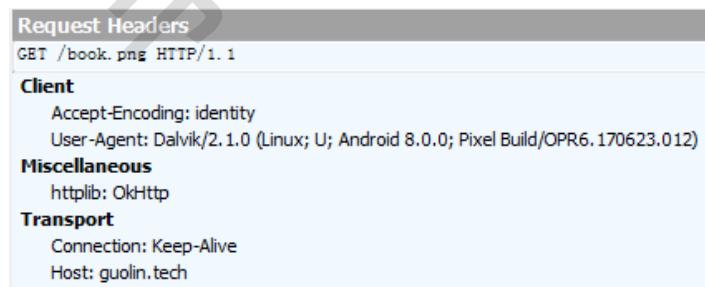
验证的方式我倒是想了很多种，比如添加OkHttp拦截器，或者自己架设一个测试用的服务器都是可以的。不过为了让大家最直接地看到验证结果，这里我准备使用Fiddler这个抓包工具来进行验证。这个工具的用法非常简单，但是限于篇幅我就不在本篇文章中介绍这个工具的用法了，还没用过这个工具的朋友们可以通过[这篇文章](#)了解一下。

在开始验证之前，我们还得要再修改一下Glide加载图片的代码才行，如下所示：

```
String url = "http://guolin.tech/book.png";
Glide.with(this)
    .load(url)
    .skipMemoryCache(true)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageView);
```

这里我把Glide的内存缓存和硬盘缓存都禁用掉了，不然的话，Glide可能会直接读取刚才缓存的图片，而不会再重新发起网络请求。

好的，现在我们重新使用Glide加载一下图片，然后观察Fiddler中的抓包情况，如下图所示。



可以看到，在HTTP请求头中确实有我们刚才自己添加的httplib: OkHttp。也就说明，Glide的HTTP通讯组件的确被替换成功了。

更简单的组件替换

上述方法是我们纯手工地将Glide的HTTP通讯组件进行了替换，如果你不想这么麻烦也是可以的，Glide官方给我们提供了非常简便的HTTP组件替换方式。并且除了支持OkHttp3之外，还支持OkHttp2和Volley。

我们只需要在gradle当中添加几行库的配置就行了。比如使用OkHttp3来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.squareup.okhttp3:okhttp:3.9.0'  
    compile 'com.github.bumptech.glide:okhttp3-  
integration:1.5.0@aar'  
}
```

使用OkHttp2来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.github.bumptech.glide:okhttp-  
integration:1.5.0@aar'  
    compile 'com.squareup.okhttp:okhttp:2.7.5'  
}
```

使用Volley来作为HTTP通讯组件的配置如下：

```
dependencies {  
    compile 'com.github.bumptech.glide:volley-  
integration:1.5.0@aar'  
    compile 'com.mcxiaoke.volley:library:1.0.19'  
}
```

当然了，这些库背后的工作原理和我们刚才自己手动实现替换HTTP组件的原理是一模一样的。而学会了手动替换组件的原理我们就能更加轻松地扩展更多丰富的功能，因此掌握这一技能还是非常重要的。

好了，那么今天的文章就到这里了。下篇文章中，我们将会利用本篇文章中学到的知识，对Glide进行一个高级的功能扩展，感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（七），实现带进度的Glide图片加载功能](#)。

3.15 Android图片加载框架最全解析（七），实现带进度的Glide图片加载功能

我们的Glide系列文章终于要进入收尾篇了。从我开始写这个系列的第一篇文章时，我就知道这会是一个很长的系列，只是没有想到竟然会写这么久。

在前面的六篇文章中，我们对Glide的方方面面都进行了学习，包括[基本用法](#)、[源码解析](#)、[缓存机制](#)、[回调与监听](#)、[图片变换](#)以及[自定义模块](#)。而今天，我们就要综合利用之前所学到的知识，来对Glide进行一个比较大的功能扩展，希望大家都已经好好阅读过了前面的六篇文章，并且有了不错的理解。

扩展目标

首先来确立一下功能扩展的目标。虽说Glide本身就已经十分强大了，但是有一个功能却长期以来都不支持，那就是监听下载进度功能。

我们都知道，使用Glide来加载一张网络上的图片是非常简单的，但是让人头疼的是，我们却无从得知当前图片的下载进度。如果这张图片很小的话，那么问题也不大，反正很快就会被加载出来。但如果这是一张比较大的GIF图，用户耐心等了很久结果图片还没显示出来，这个时候你就会觉得下载进度功能是十分有必要的了。

好的，那么我们今天的目标就是对Glide进行功能扩展，使其支持监听图片下载进度的功能。

开始

今天这篇文章我会带着大家从零去创建一个新的项目，一步步地进行实现，最终完成一个带进度的Glide图片加载的Demo。当然，在本篇文章的最后我会提供这个Demo的完整源码，但是这里我仍然希望大家能用心跟着我一步步来编写。

那么我们现在就开始吧，首先创建一个新项目，就叫做GlideProgressTest吧。

项目创建完成后的第一件事就是要将必要的依赖库引入到当前的项目当中，目前我们必须要依赖的两个库就是Glide和OkHttp。在app/build.gradle文件当中添加如下配置：

```
dependencies {  
    compile 'com.github.bumptech.glide:glide:3.7.0'  
    compile 'com.squareup.okhttp3:okhttp:3.9.0'  
}
```

另外，由于Glide和OkHttp都需要用到网络功能，因此我们还得在AndroidManifest.xml中声明一下网络权限才行：

```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

好了，这样准备工作就完成了。

替换通讯组件

通过[第二篇文章](#)的源码分析，我们知道了Glide内部HTTP通讯组件的底层实现是基于HttpURLConnection来进行定制的。但是HttpURLConnection的可扩展性比较有限，我们在它的基础之上无法实现监听下载进度的功能，因此今天的第一大动作就是要将Glide中的HTTP通讯组件替换成OkHttp。

关于HTTP通讯组件的替换原理和替换方式，我在[第六篇文章](#)当中都介绍得比较清楚了，这里就不再赘述。下面我们就来开始快速地替换一下。

新建一个OkHttpFetcher类，并且实现DataFetcher接口，代码如下所示：

```
public class OkHttpFetcher implements  
DataFetcher<InputStream> {  
  
    private final OkHttpClient client;  
    private final GlideUrl url;  
    private InputStream stream;  
    private ResponseBody responseBody;  
    private volatile boolean isCancelled;
```

```
    public okhttpFetcher(okHttpClient client, GlideUrl url) {
        this.client = client;
        this.url = url;
    }

    @Override
    public InputStream loadData(Priority priority) throws
Exception {
        Request.Builder requestBuilder = new
Request.Builder()
        .url(url.toStringUrl());
        for (Map.Entry<String, String> headerEntry :
url.getHeaders().entrySet()) {
            String key = headerEntry.getKey();
            requestBuilder.addHeader(key,
headerEntry.getValue());
        }
        Request request = requestBuilder.build();
        if (isCancelled) {
            return null;
        }
        Response response =
client.newCall(request).execute();
        responseBody = response.body();
        if (!response.isSuccessful() || responseBody ==
null) {
            throw new IOException("Request failed with
code: " + response.code());
        }
        stream =
ContentLengthInputStream.obtain(responseBody.byteStream()
,
                    responseBody.contentLength());
        return stream;
    }
}
```

```
@Override  
public void cleanup() {  
    try {  
        if (stream != null) {  
            stream.close();  
        }  
        if (responseBody != null) {  
            responseBody.close();  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

```
@Override  
public String getId() {  
    return url.getCacheKey();  
}
```

```
@Override  
public void cancel() {  
    isCancelled = true;  
}  
}
```

然后新建一个OkHttpGlideUrlLoader类，并且实现ModelLoader

```
public class OkHttpGlideUrlLoader implements  
ModelLoader<.GlideUrl, InputStream> {  
  
    private OkHttpClient okHttpClient;  
  
    public static class Factory implements  
ModelLoaderFactory<.GlideUrl, InputStream> {  
  
        private OkHttpClient client;  
  
        public Factory() {
```

```
    }

    public Factory(OkHttpClient client) {
        this.client = client;
    }

    private synchronized OkHttpClient
getOkHttpClient() {
        if (client == null) {
            client = new OkHttpClient();
        }
        return client;
    }

    @Override
    public ModelLoader<GlideUrl, InputStream>
build(Context context, GenericLoaderFactory factories) {
        return new
OkHttpGlideUrlLoader(getOkHttpClient());
    }

    @Override
    public void teardown() {
    }

    public OkHttpGlideUrlLoader(OkHttpClient client) {
        this.okHttpClient = client;
    }

    @Override
    public DataFetcher<InputStream>
getResourceFetcher(GlideUrl model, int width, int height)
{
        return new OkHttpFetcher(okHttpClient, model);
    }
}
```

接下来，新建一个MyGlideModule类并实现GlideModule接口，然后在registerComponents()方法中将我们刚刚创建的OkHttpGlideUrlLoader和OkHttpFetcher注册到Glide当中，将原来的HTTP通讯组件给替换掉，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
    glide) {
        glide.register(GlideUrl.class, InputStream.class,
        new OkHttpGlideUrlLoader.Factory());
    }
}
```

最后，为了让Glide能够识别我们自定义的MyGlideModule，还得在AndroidManifest.xml文件当中加入如下配置才行：

```
<manifest>
    ...
    <application>
        <meta-data
            android:name="com.example.glideprogressstest.MyGlideModule"
            android:value="GlideModule" />
        ...
    </application>
</manifest>
```

OK，这样我们就把Glide中的HTTP通讯组件成功替换成OkHttp了。

实现下载进度监听

那么，将HTTP通讯组件替换成OkHttp之后，我们又该如何去实现监听下载进度的功能呢？这就要依靠OkHttp强大的拦截器机制了。

我们只要向OkHttp中添加一个自定义的拦截器，就可以在拦截器中捕获到整个HTTP的通讯过程，然后加入一些自己的逻辑来计算下载进度，这样就可以实现下载进度监听的功能了。

拦截器属于OkHttp的高级功能，不过即使你之前并没有接触过拦截器，我相信你也能轻松看懂本篇文章的，因为它本身并不难。

确定了实现思路之后，那我们就开始动手吧。首先创建一个没有任何逻辑的空拦截器，新建ProgressInterceptor类并实现Interceptor接口，代码如下所示：

```
public class ProgressInterceptor implements Interceptor {  
  
    @Override  
    public Response intercept(Chain chain) throws  
    IOException {  
        Request request = chain.request();  
        Response response = chain.proceed(request);  
        return response;  
    }  
}
```

这个拦截器中我们可以说是什么都没有做。就是拦截到了OkHttp的请求，然后调用proceed()方法去处理这个请求，最终将服务器响应的Response返回。

接下来我们需要启用这个拦截器，修改MyGlideModule中的代码，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
        OkHttpClient.Builder builder = new
OkHttpClient.Builder();
        builder.addInterceptor(new
ProgressInterceptor());
        OkHttpClient okHttpClient = builder.build();
        glide.register(GlideUrl.class, InputStream.class,
new OkHttpGlideUrlLoader.Factory(okHttpClient));
    }
}
```

这里我们创建了一个OkHttpClient.Builder，然后调用addInterceptor()方法将刚才创建的ProgressInterceptor添加进去，最后将构建出来的新 OkHttpClient对象传入到OkHttpGlideUrlLoader.Factory中即可。

好的，现在自定义的拦截器已经启用了，接下来就可以开始去实现下载进度监听的具体逻辑了。首先新建一个ProgressListener接口，用于作为进度监听回调的工具，如下所示：

```
public interface ProgressListener {
    void onProgress(int progress);
}
```

然后我们在ProgressInterceptor中加入注册下载监听和取消注册下载监听的方法。修改ProgressInterceptor中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {
```

```
    static final Map<String, ProgressListener>
LISTENER_MAP = new HashMap<>();

    public static void addListener(String url,
ProgressListener listener) {
        LISTENER_MAP.put(url, listener);
    }

    public static void removeListener(String url) {
        LISTENER_MAP.remove(url);
    }

    @Override
    public Response intercept(Chain chain) throws
IOException {
        Request request = chain.request();
        Response response = chain.proceed(request);
        return response;
    }
}
```

可以看到，这里使用了一个Map来保存注册的监听器，Map的键是一个URL地址。之所以要这么做，是因为你可能会使用Glide同时加载很多张图片，而这种情况下，必须要能区分出来每个下载进度的回调到底是对哪个图片URL地址的。

接下来就要到今天最复杂的部分了，也就是下载进度的具体计算。我们需要新建一个ProgressResponseBody类，并让它继承自OkHttp的ResponseBody，然后在这个类当中去编写具体的监听下载进度的逻辑，代码如下所示：

```
public class ProgressResponseBody extendsResponseBody {

    private static final String TAG =
"ProgressResponseBody";
```

```
private BufferedSource bufferedSource;

private ResponseBody responseBody;

private ProgressListener listener;

public ProgressResponseBody(String url, ResponseBody
responseBody) {
    this.responseBody = responseBody;
    listener =
ProgressInterceptor.LISTENER_MAP.get(url);
}

@Override
public MediaType contentType() {
    return responseBody.contentType();
}

@Override
public long contentLength() {
    return responseBody.contentLength();
}

@Override
public BufferedSource source() {
    if (bufferedSource == null) {
        bufferedSource = Okio.buffer(new
ProgressSource(responseBody.source()));
    }
    return bufferedSource;
}

private class ProgressSource extends ForwardingSource
{

    long totalBytesRead = 0;
```

```
int currentProgress;

ProgressSource(Source source) {
    super(source);
}

@Override
public long read(Buffer sink, long byteCount)
throws IOException {
    long bytesRead = super.read(sink, byteCount);
    long fullLength =
responseBody.contentLength();
    if (bytesRead == -1) {
        totalBytesRead = fullLength;
    } else {
        totalBytesRead += bytesRead;
    }
    int progress = (int) (100f * totalBytesRead /
fullLength);
    Log.d(TAG, "download progress is " +
progress);
    if (listener != null && progress !=
currentProgress) {
        listener.onProgress(progress);
    }
    if (listener != null && totalBytesRead ==
fullLength) {
        listener = null;
    }
    currentProgress = progress;
    return bytesRead;
}
}
```

其实这段代码也不是很难，下面我来简单解释一下。首先，我们定义了一个ProgressResponseBody的构造方法，该构造方法中要求传入一个url参数和一个ResponseBody参数。那么很显然，url参数就是图片的url地址了，而ResponseBody参数则是OkHttp拦截到的原始的ResponseBody对象。然后在构造方法中，我们调用了ProgressInterceptor中的LISTENER_MAP来去获取该url对应的监听器回调对象，有了这个对象，待会就可以回调计算出来的下载进度了。

由于继承了ResponseBody类之后一定要重写contentType()、contentLength()和source()这三个方法，我们在contentType()和contentLength()方法中直接就调用传入的原始ResponseBody的contentType()和contentLength()方法即可，这相当于一种委托模式。但是在source()方法中，我们就必须加入点自己的逻辑了，因为这里要涉及到具体的下载进度计算。

那么我们具体看一下source()方法，这里先是调用了原始ResponseBody的source()方法来去获取Source对象，接下来将这个Source对象封装到了一个ProgressSource对象当中，最终再用Okio的buffer()方法封装成BufferedSource对象返回。

那么这个ProgressSource是什么呢？它是一个我们自定义的继承自ForwardingSource的实现类。ForwardingSource也是一个使用委托模式的工具，它不处理任何具体的逻辑，只是负责将传入的原始Source对象进行中转。但是，我们使用ProgressSource继承自ForwardingSource，那么就可以在中转的过程中加入自己的逻辑了。

可以看到，在ProgressSource中我们重写了read()方法，然后在read()方法中获取该次读取到的字节数以及下载文件的总字节数，并进行一些简单的数学计算就能算出当前的下载进度了。这里我先使用Log工具将算出的结果打印了一下，再通过前面获取到的回调监听器对象将结果进行回调。

好的，现在计算下载进度的逻辑已经完成了，那么我们快点在拦截器当中使用它吧。修改ProgressInterceptor中的代码，如下所示：

```
public class ProgressInterceptor implements Interceptor {  
    ...  
}
```

```
    @Override
    public Response intercept(Chain chain) throws
    IOException {
        Request request = chain.request();
        Response response = chain.proceed(request);
        String url = request.url().toString();
       ResponseBody body = response.body();
        Response newResponse =
        response.newBuilder().body(new ProgressResponseBody(url,
        body)).build();
        return newResponse;
    }

}
```

这里也都是一些OkHttp的简单用法。我们通过Response的newBuilder()方法来创建一个新的Response对象，并把它的body替换成刚才实现的ProgressResponseBody，最终将新的Response对象进行返回，这样计算下载进度的逻辑就能生效了。

代码写到这里，我们就可以来运行一下程序了。现在无论是加载任何网络上的图片，都应该是可以监听到它的下载进度的。

修改activity_main.xml中的代码，如下所示：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Image"
        android:onClick="loadImage"
```

```
    />

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

很简单，这里使用了一个Button按钮来加载图片，使用了一个ImageView来展示图片。

然后修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

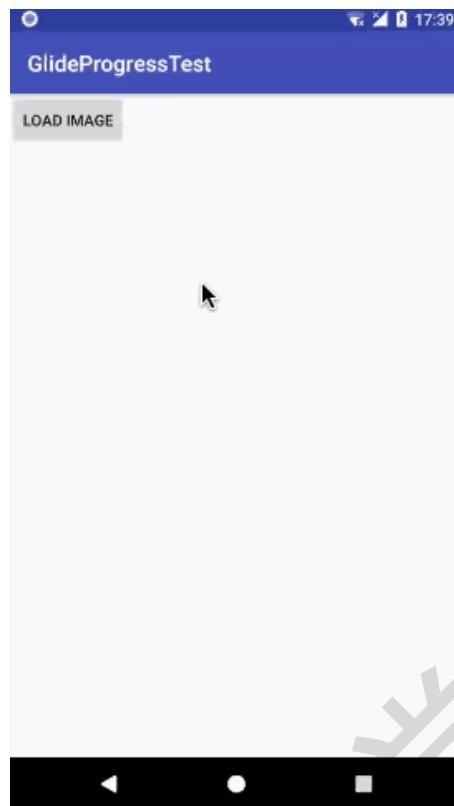
    String url = "http://guolin.tech/book.png";

    ImageView image;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.image);
    }

    public void loadImage(View view) {
        Glide.with(this)
            .load(url)
            .diskCacheStrategy(DiskCacheStrategy.NONE)
            .override(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL)
            .into(image);
    }
}
```

现在就可以运行一下程序了，效果如下图所示。



OK，图片已经加载出来了。那么怎么验证有没有成功监听到图片的下载进度呢？还记得我们刚才在ProgressResponseBody中加的打印日志吗？现在只要去logcat中观察一下就知道了，如下图所示：

```
D/ProgressResponseBody: download progress is 0
D/ProgressResponseBody: download progress is 6
D/ProgressResponseBody: download progress is 8
D/ProgressResponseBody: download progress is 9
D/ProgressResponseBody: download progress is 10
D/ProgressResponseBody: download progress is 11
D/ProgressResponseBody: download progress is 12
D/ProgressResponseBody: download progress is 13
D/ProgressResponseBody: download progress is 14
D/ProgressResponseBody: download progress is 15
D/ProgressResponseBody: download progress is 16
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 17
D/ProgressResponseBody: download progress is 18
D/ProgressResponseBody: download progress is 19
D/ProgressResponseBody: download progress is 20
D/ProgressResponseBody: download progress is 21
D/ProgressResponseBody: download progress is 22
D/ProgressResponseBody: download progress is 23
D/ProgressResponseBody: download progress is 24
D/ProgressResponseBody: download progress is 25
D/ProgressResponseBody: download progress is 26
D/ProgressResponseBody: download progress is 27
```

由此可见，下载进度监听功能已经成功实现了。

进度显示

虽然现在我们已经能够监听到图片的下载进度了，但是这个进度目前还只能显示在控制台打印当中，这对于用户来说是没有任何意义的，因此我们下一步就是要想办法将下载进度显示到界面上。

现在修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    String url = "http://guolin.tech/book.png";

    ImageView image;

    ProgressDialog progressDialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        image = (ImageView) findViewById(R.id.image);
        progressDialog = new ProgressDialog(this);

        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progressDialog.setMessage("加载中");
    }

    public void loadImage(View view) {
        ProgressInterceptor.addListener(url, new
ProgressListener() {
            @Override
            public void onProgress(int progress) {
                progressDialog.setProgress(progress);
            }
        });
        Glide.with(this)
            .load(url)
            .diskCacheStrategy(DiskCacheStrategy.NONE)
```

```
        .override(Target.SIZE_ORIGINAL,
Target.SIZE_ORIGINAL)
        .into(new
GlideDrawableImageViewTarget(image) {
    @Override
    public void onLoadStarted(Drawable
placeholder) {
        super.onLoadStarted(placeholder);
        progressDialog.show();
    }

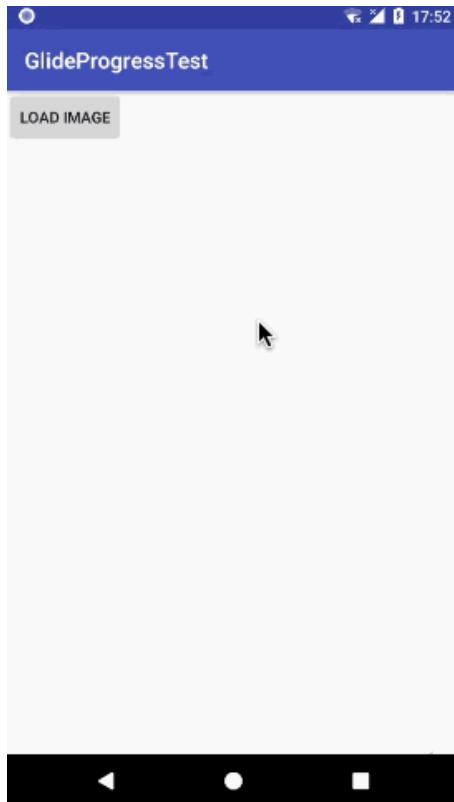
    @Override
    public void
onResourceReady(GlideDrawable resource, GlideAnimation<?
super GlideDrawable> animation) {
        super.onResourceReady(resource,
animation);
        progressDialog.dismiss();

ProgressInterceptor.removeListener(url);
    }
});
}
```

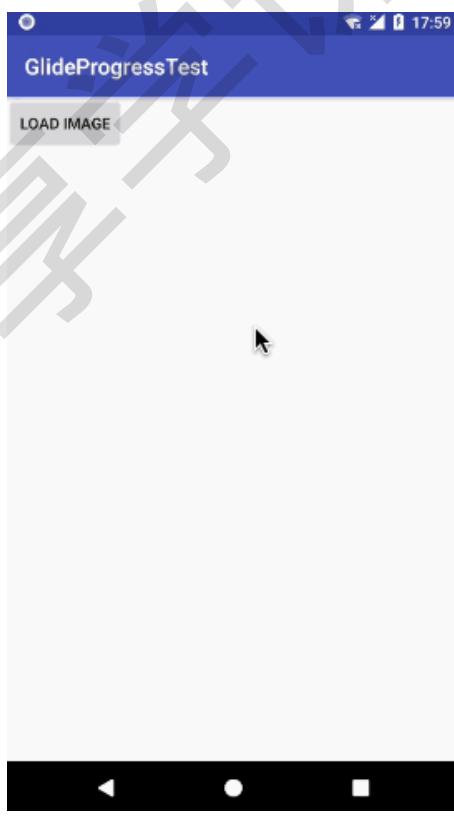
代码并不复杂。这里我们新增了一个ProgressDialog用来显示下载进度，然后在loadImage()方法中，调用了ProgressInterceptor.addListener()方法来去注册一个下载监听器，并在onProgress()回调方法中更新当前的下载进度。

最后，Glide的into()方法也做了修改，这次是into到了一个GlideDrawableImageViewTarget当中。我们重写了它的onLoadStarted()方法和onResourceReady()方法，从而实现当图片开始加载的时候显示进度对话框，当图片加载完成时关闭进度对话框的功能。

现在重新运行一下程序，效果如下图所示。



当然，不仅仅是静态图片，体积比较大的GIF图也是可以成功监听到下载进度的。比如我们把图片的url地址换成<http://guolin.tech/test.gif>，重新运行程序，效果如下图所示。



好了，这样我们就把带进度的Glide图片加载功能完整地实现了一遍。虽然这个例子当中的界面都比较粗糙，下载进度框也是使用的最简陋的，不过只要将功能学会了，界面那都不是事，大家后期可以自己进行各种界面优化。

最后，如果你想要下载完整的Demo，请[点击这里](#)。

写了大半年的一个系列就那么要结束了，突然还有一点点小不舍。如果大家能将整个系列的七篇文章都很好地掌握了，那么现在自称Glide高手应该不算过分。

其实在刚打算写这个系列的时候，我是准备写八篇文章，结果最后满打满算就只写出了七篇。那么为了兑现自己当初八篇的承诺，我准备最后一篇写一下关于Glide 4.0版本的用法，顺便让我自己也找个契机去研究一下新版本。当然，这并不是说Glide 3.7版本就已经淘汰了，事实上，Glide 3.7版本十分稳定，而且还能几乎完全满足我平时开发的所有需求，是可以长期使用下去的一个版本。

感兴趣的朋友请继续阅读 [Android图片加载框架最全解析（八），带你全面了解Glide 4的用法](#)。

3.16 Android图片加载框架最全解析（八），带你全面了解Glide 4的用法

本篇将是我们这个Glide系列的最后一篇文章。

其实在写这个系列第一篇文章的时候，Glide就推出4.0.0的RC版了。那个时候因为我一直研究的都是Glide 3.7.0版本，再加上RC版本还不太稳定，因此整个系列也都是基于3.7.0版本来写的。

而现在，Glide的最新版本已经出到了4.4.0，可以说Glide 4已经是相当成熟和稳定了。而且也不断有朋友一直在留言，想让我讲一讲Glide 4的用法，因为Glide 4相对于Glide 3改动貌似还是挺大的，学完了Glide 3再去使用Glide 4，发现根本就无法使用。

OK，那么今天就让我们用《带你全面了解Glide 4的用法》这样一篇文章，给这个Glide系列画上一个圆满的句号。

Glide 4概述

刚才有说到，有些朋友觉得Glide 4相对于Glide 3改动非常大，其实不然。之所以大家会有这种错觉，是因为你将Glide 3的用法直接搬到Glide 4中去使用，结果IDE全面报错，然后大家可能就觉得Glide 4的用法完全变掉了。

其实Glide 4相对于Glide 3的变动并不大，只是你还没有了解它的变动规则而已。一旦你掌握了Glide 4的变动规则之后，你会发现大多数Glide 3的用法放到Glide 4上都还是通用的。

我对Glide 4进行了一个大概的研究之后，发现Glide 4并不能算是有什么突破性的升级，而更多是一些API工整方面的优化。相比于Glide 3的API，Glide 4进行了更加科学合理地调整，使得易读性、易写性、可扩展性等方面都有了不错的提升。但如果你已经对Glide 3非常熟悉的话，并不是就必须要切换到Glide 4上来，因为Glide 4上能实现的功能Glide 3也都能实现，而且Glide 4在性能方面也并没有什么提升。

但是对于新接触Glide的朋友而言，那就没必要再去学习Glide 3了，直接上手Glide 4就是最佳的选择了。

好了，对Glide 4进行一个基本的概述之后，接下来我们就要正式开始学习它的用法了。刚才我已经说了，Glide 4的用法相对于Glide 3其实改动并不大。在前面的七篇文章中，我们已经学习了Glide 3的基本用法、缓存机制、回调与监听、图片变换、自定义模块等用法，那么今天这篇文章的目标就很简单了，就是要掌握如何在Glide 4上实现之前所学习过的所有功能，那么我们现在就开始吧。

开始

要想使用Glide，首先需要将这个库引入到我们的项目当中。新建一个Glide4Test项目，然后在app/build.gradle文件当中添加如下依赖：

```
dependencies {
    implementation 'com.github.bumptech.glide:glide:4.4.0'
    annotationProcessor 'com.github.bumptech.glide:compiler:4.4.0'
}
```

注意，相比于Glide 3，这里要多添加一个compiler的库，这个库是用于生成Generated API的，待会我们会讲到它。

另外，Glide中需要用到网络功能，因此你还得在AndroidManifest.xml中声明一下网络权限才行：

```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

就是这么简单，然后我们就可以自由地使用Glide中的任意功能了。

加载图片

现在我们就来尝试一下如何使用Glide来加载图片吧。比如这是一张图片的地址：

```
http://guolin.tech/book.png
```

然后我们想要在程序当中去加载这张图片。

那么首先打开项目的布局文件，在布局当中加入一个Button和一个ImageView，如下所示：

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:orientation="vertical">  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Load Image"  
        android:onClick="loadImage"  
    />  
  
    <ImageView  
        android:id="@+id/image_view"  
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent" />  
  
    </LinearLayout>
```

为了让用户点击Button的时候能够将刚才的图片显示在ImageView上，我们需要修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {  
  
    ImageView imageView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        imageView = (ImageView)  
findviewById(R.id.image_view);  
    }  
  
    public void loadImage(View view) {  
        String url = "http://guolin.tech/book.png";  
        Glide.with(this).load(url).into(imageView);  
    }  
}
```

没错，就是这么简单。现在我们来运行一下程序，效果如下图所示：



可以看到，一张网络上的图片已经被成功下载，并且展示到ImageView上了。

你会发现，到目前为止，Glide 4的用法和Glide 3是完全一样的，实际上核心的代码就只有这一行而已：

```
Glide.with(this).load(url).into(imageview);
```

仍然还是传统的三步走：先with()，再load()，最后into()。对这行代码的解读，我在[Android图片加载框架最全解析（一），Glide的基本用法](#)这篇文章中讲解的很清楚了，这里就不再赘述。

好了，现在你已经成功入门Glide 4了，那么接下来就让我们学习一下Glide 4的更多用法吧。

占位图

观察刚才加载网络图片的效果，你会发现，点击了Load Image按钮之后，要稍微等一会图片才会显示出来。这其实很容易理解，因为从网络上下载图片本来就是需要时间的。那么我们有没有办法再优化一下用户体验呢？当然可以，Glide提供了各种各样非常丰富的API支持，其中就包括了

占位图功能。

顾名思义，占位图就是指在图片的加载过程中，我们先显示一张临时的图片，等图片加载出来了再替换成要加载的图片。

下面我们就来学习一下Glide占位图功能的使用方法，首先我事先准备好了一张loading.jpg图片，用来作为占位图显示。然后修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.loading);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

没错，就是这么简单。这里我们先创建了一个RequestOptions对象，然后调用它的placeholder()方法来指定占位图，再将占位图片的资源id传入到这个方法中。最后，在Glide的三步走之间加入一个apply()方法，来应用我们刚才创建的RequestOptions对象。

不过如果你现在重新运行一下代码并点击Load Image，很可能是根本看不到占位图效果的。因为Glide有非常强大的缓存机制，我们刚才加载图片的时候Glide自动就已经将它缓存下来了，下次加载的时候将会直接从缓存中读取，不会再去找网络下载了，因而加载的速度非常快，所以占位图可能根本来不及显示。

因此这里我们还需要稍微做一点修改，来让占位图能有机会显示出来，修改代码如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.loading)
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

可以看到，这里在`RequestOptions`对象中又串接了一个`diskCacheStrategy()`方法，并传入`DiskCacheStrategy.NONE`参数，这样就可以禁用掉Glide的缓存功能。

关于Glide缓存方面的内容我们待会儿会进行更详细的讲解，这里只是为了测试占位图功能而加的一个额外配置，暂时你只需要知道禁用缓存必须这么写就可以了。

现在重新运行一下代码，效果如下图所示：



可以看到，当点击Load Image按钮之后会立即显示一张占位图，然后等真正的图片加载完成之后会将占位图替换掉。

除了这种加载占位图之外，还有一种异常占位图。异常占位图就是指，如果因为某些异常情况导致图片加载失败，比如说手机网络信号不好，这个时候就显示这张异常占位图。

异常占位图的用法相信你已经可以猜到了，首先准备一张`error.jpg`图片，然后修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()
    .placeholder(R.drawable.ic_launcher_background)
    .error(R.drawable.error)
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageView);
```

很简单，这里又串接了一个error()方法就可以指定异常占位图了。

其实看到这里，如果你熟悉Glide 3的话，相信你已经掌握Glide 4的变化规律了。在Glide 3当中，像placeholder()、error()、diskCacheStrategy()等等一系列的API，都是直接串联在Glide三步走方法中使用的。

而Glide 4中引入了一个RequestOptions对象，将这一系列的API都移动到了RequestOptions当中。这样做好处是可以使我们摆脱冗长的Glide加载语句，而且还能进行自己的API封装，因为RequestOptions是可以作为参数传入到方法中的。

比如你就可以写出这样的Glide加载工具类：

```
public class GlideUtil {

    public static void load(Context context,
                           String url,
                           ImageView imageView,
                           RequestOptions options) {
        Glide.with(context)
            .load(url)
            .apply(options)
            .into(imageView);
    }

}
```

指定图片大小

实际上，使用Glide在大多数情况下我们都是不需要指定图片大小的，因为Glide会自动根据ImageView的大小来决定图片的大小，以此保证图片不会占用过多的内存从而引发OOM。

不过，如果你真的有这样的需求，必须给图片指定一个固定的大小，Glide仍然是支持这个功能的。修改Glide加载部分的代码，如下所示：

```
RequestOptions options = new RequestOptions()
    .override(200, 100);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

仍然非常简单，这里使用override()方法指定了一个图片的尺寸。也就是说，Glide现在只会将图片加载成200*100像素的尺寸，而不会管你的ImageView的大小是多少了。

如果你想加载一张图片的原始尺寸的话，可以使用Target.SIZE_ORIGINAL关键字，如下所示：

```
RequestOptions options = new RequestOptions()
    .override(Target.SIZE_ORIGINAL);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

这样的话，Glide就不会再去自动压缩图片，而是会去加载图片的原始尺寸。当然，这种写法也会面临着更高的OOM风险。

缓存机制

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就来分别学习一下这两种缓存的使用方法。

首先来看内存缓存。

你要知道，默认情况下，Glide自动就是开启内存缓存的。也就是说，当我们使用Glide加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用Glide再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个RecyclerView当中反复上下滑动，RecyclerView中只要是Glide加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而Glide最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为Glide默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide对此提供了接口：

```
RequestOptions options = new RequestOptions()
    .skipMemoryCache(true);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageView);
```

可以看到，只需要调用skipMemoryCache()方法并传入true，就表示禁用掉Glide的内存缓存功能。

接下来我们开始学习硬盘缓存方面的内容。

其实在刚刚学习占位图功能的时候，我们就使用过硬盘缓存的功能了。当时为了禁止Glide对图片进行硬盘缓存而使用了如下代码：

```
RequestOptions options = new RequestOptions()
    .diskCacheStrategy(DiskCacheStrategy.NONE);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

调用diskCacheStrategy()方法并传入DiskCacheStrategy.NONE，就可以禁用掉Glide的硬盘缓存功能了。

这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收五种参数：

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.DATA： 表示只缓存原始图片。
- DiskCacheStrategy.RESOURCE： 表示只缓存转换过后的图片。
- DiskCacheStrategy.ALL： 表示既缓存原始图片，也缓存转换过后的图片。
- DiskCacheStrategy.AUTOMATIC： 表示让Glide根据图片资源智能地选择使用哪一种缓存策略（默认选项）。

其中，DiskCacheStrategy.DATA对应Glide 3中的DiskCacheStrategy.SOURCE，DiskCacheStrategy.RESOURCE对应Glide 3中的DiskCacheStrategy.RESULT。而DiskCacheStrategy.AUTOMATIC是Glide 4中新增的一种缓存策略，并且在不指定diskCacheStrategy的情况下默认使用就是的这种缓存策略。

上面五种参数的解释本身并没有什么难理解的地方，但是关于转换过后的图片这个概念大家可能需要了解一下。就是当我们使用Glide去加载一张图片的时候，Glide默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在稍后学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。

好的，关于Glide 4硬盘缓存的内容就讲到这里。想要了解更多Glide缓存方面的知识，可以参考[Android图片加载框架最全解析（三），深入探究Glide的缓存机制](#)这篇文章。

指定加载格式

我们都知道，Glide其中一个非常亮眼的功能就是可以加载GIF图片，而同样作为非常出色的图片加载框架的Picasso是不支持这个功能的。

而且使用Glide加载GIF图并不需要编写什么额外的代码，Glide内部会自动判断图片格式。比如我们将加载图片的URL地址改成一张GIF图，如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/test.gif")
    .into(imageview);
```

现在重新运行一下代码，效果如下图所示：



也就是说，不管我们传入的是一张普通图片，还是一张GIF图片，Glide都会自动进行判断，并且可以正确地把它解析并展示出来。

但是如果我想指定加载格式该怎么办呢？就比如说，我希望加载的这张图必须是一张静态图片，我不需要Glide自动帮我判断它到底是静图还是GIF图。

想实现这个功能仍然非常简单，我们只需要再串接一个新的方法就可以了，如下所示：

```
Glide.with(this)
    .asBitmap()
    .load("http://guolin.tech/test.gif")
    .into(imageview);
```

可以看到，这里在with()方法的后面加入了一个asBitmap()方法，这个方法的意思就是说这里只允许加载静态图片，不需要Glide去帮我们自动进行图片格式的判断了。如果你传入的还是一张GIF图的话，Glide会展示这张GIF图的第一帧，而不会去播放它。

熟悉Glide 3的朋友对asBitmap()方法肯定不会陌生对吧？但是千万不要觉得这里就没有陷阱了，在Glide 3中的语法是先load()再asBitmap()的，而在Glide 4中是先asBitmap()再load()的。乍一看可能分辨不出来有什么区别，但如果你写错了顺序就肯定会报错了。

那么类似地，既然我们能强制指定加载静态图片，就也能强制指定加载动态图片，对应的方法是asGif()。而Glide 4中又新增了asFile()方法和asDrawable()方法，分别用于强制指定文件格式的加载和Drawable格式的加载，用法都比较简单，就不再进行演示了。

回调与监听

回调与监听这部分的内容稍微有点多，我们分成四部分来学习一下。

1. into()方法

我们都应该知道Glide的into()方法中是可以传入ImageView的。那么into()方法还可以传入别的参数吗？我们可以让Glide加载出来的图片不显示到ImageView上吗？答案是肯定的，这就需要用到自定义Target功能。

Glide中的Target功能多样且复杂，下面我就先简单演示一种SimpleTarget的用法吧，代码如下所示：

```
simpleTarget<Drawable> simpleTarget = new
simpleTarget<Drawable>() {
    @Override
    public void onResourceReady(Drawable resource,
Transition<? super Drawable> transition) {
        imageView.setImageDrawable(resource);
    }
};

public void loadImage(View view) {
    Glide.with(this)
        .load("http://guolin.tech/book.png")
        .into(simpleTarget);
}
```

这里我们创建了一个SimpleTarget的实例，并且指定它的泛型是Drawable，然后重写了onResourceReady()方法。在onResourceReady()方法中，我们就可以获取到Glide加载出来的图片对象了，也就是方法参数中传过来的Drawable对象。有了这个对象之后你可以使用它进行任意的逻辑操作，这里我只是简单地把它显示到了ImageView上。

SimpleTarget的实现创建好了，那么只需要在加载图片的时候将它传入到into()方法中就可以了。

这里限于篇幅原因我只演示了自定义Target的简单用法，想学习更多相关的内容可以去阅读 [Android图片加载框架最全解析（四），玩转Glide的回调与监听](#)。

2. preload()方法

Glide加载图片虽说非常智能，它会自动判断该图片是否已经有缓存了，如果说有的话就直接从缓存中读取，没有的话再从网络去下载。但是如果我希望提前对图片进行一个预加载，等真正需要加载图片的时候就直接从缓存中读取，不想再等待漫长的网络加载时间了，这该怎么办呢？

不用担心，Glide专门给我们提供了预加载的接口，也就是preload()方法，我们只需要直接使用就可以了。

preload()方法有两个方法重载，一个不带参数，表示将会加载图片的原始尺寸，另一个可以通过参数指定加载图片的宽和高。

preload()方法的用法也非常简单，直接使用它来替换into()方法即可，如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .preload();
```

调用了预加载之后，我们以后想再去加载这张图片就会非常快了，因为Glide会直接从缓存当中去读取图片并显示出来，代码如下所示：

```
Glide.with(this)
    .load("http://guolin.tech/book.png")
    .into(imageview);
```

3. submit()方法

一直以来，我们使用Glide都是为了将图片显示到界面上。虽然我们知道Glide会在图片的加载过程中对图片进行缓存，但是缓存文件到底是存在哪里的，以及如何去直接访问这些缓存文件？我们都还不知道。

其实Glide将图片加载接口设计成这样也是希望我们使用起来更加的方便，不用过多去考虑底层的实现细节。但如果我现在就是想要去访问图片的缓存文件该怎么办呢？这就需要用到submit()方法了。

submit()方法其实就是对应的Glide 3中的downloadOnly()方法，和preload()方法类似，submit()方法也是可以替换into()方法的，不过submit()方法的用法明显要比preload()方法复杂不少。这个方法只会下载图片，而不会对图片进行加载。当图片下载完成之后，我们可以得到图片的存储路径，以便后续进行操作。

那么首先我们还是先来看下基本用法。submit()方法有两个方法重载：

- submit()
- submit(int width, int height)

其中submit()方法是用于下载原始尺寸的图片，而submit(int width, int height)则可以指定下载图片的尺寸。

这里就以submit()方法来举例。当调用了submit()方法后会立即返回一个FutureTarget对象，然后Glide会在后台开始下载图片文件。接下来我们调用FutureTarget的get()方法就可以去获取下载好的图片文件了，如果此时图片还没有下载完，那么get()方法就会阻塞住，一直等到图片下载完成才会有值返回。

下面我们通过一个例子来演示一下吧，代码如下所示：

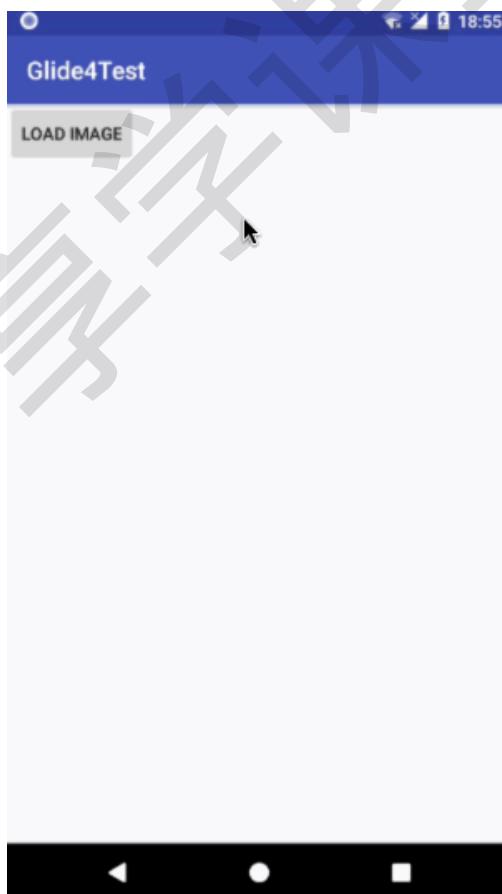
```
public void downloadImage() {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                String url =  
                    "http://www.guolin.tech/book.png";  
                final Context context =  
                    getApplicationContext();  
                FutureTarget<File> target =  
                    Glide.with(context)  
                        .asFile()  
                        .load(url)  
                        .submit();  
                final File imageFile = target.get();  
                runOnUiThread(new Runnable() {  
                    @Override  
                    public void run() {  
                        Toast.makeText(context,  
                            imageFile.getPath(), Toast.LENGTH_LONG).show();  
                    }  
                });  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }).start();  
}
```

这段代码稍微有一点点长，我带着大家解读一下。首先，submit()方法必须用在子线程当中，因为刚才说了FutureTarget的get()方法是会阻塞线程的，因此这里的一步就是new了一个Thread。在子线程当中，我们先获取了一个Application Context，这个时候不能再用Activity作为Context了，因为会有Activity销毁了但子线程还没执行完这种可能出现。

接下来就是Glide的基本用法，只不过将into()方法替换成submit()方法，并且还使用了一个asFile()方法来指定加载格式。submit()方法会返回一个FutureTarget对象，这个时候其实Glide已经开始在后台下载图片了，我们随时都可以调用FutureTarget的get()方法来获取下载的图片文件，只不过如果图片还没下载好线程会暂时阻塞住，等下载完成了才会把图片的File对象返回。

最后，我们使用runOnUiThread()切回到主线程，然后使用Toast将下载好的图片文件路径显示出来。

现在重新运行一下代码，效果如下图所示。



这样我们就能清晰地看出来图片完整的缓存路径是什么了。

4. listener()方法

其实listener()方法的作用非常普遍，它可以用来监听Glide加载图片的状态。举个例子，比如说我们刚才使用了preload()方法来对图片进行预加载，但是我怎样确定预加载有没有完成呢？还有如果Glide加载图片失败了，我该怎样调试错误的原因呢？答案都在listener()方法当中。

下面来看下listener()方法的基本用法吧，不同于刚才几个方法都是要替换into()方法的，listener()是结合into()方法一起使用的，当然也可以结合preload()方法一起使用。最基本的用法如下所示：

```
Glide.with(this)
    .load("http://www.guolin.tech/book.png")
    .listener(new RequestListener<Drawable>() {
        @Override
        public boolean onLoadFailed(@Nullable
GlideException e, Object model, Target<Drawable> target,
boolean isFirstResource) {
            return false;
        }

        @Override
        public boolean onResourceReady(Drawable
resource, Object model, Target<Drawable> target,
DataSource dataSource, boolean isFirstResource) {
            return false;
        }
    })
    .into(imageview);
```

这里我们在into()方法之前串接了一个listener()方法，然后实现了一个RequestListener的实例。其中RequestListener需要实现两个方法，一个onResourceReady()方法，一个onLoadFailed()方法。从方法名上就可以看出来了，当图片加载完成的时候就会回调onResourceReady()方法，而当图片加载失败的时候就会回调onLoadFailed()方法，onLoadFailed()方法中会将失败的GlideException参数传进来，这样我们就可以定位具体失败的原因了。

没错，`listener()`方法就是这么简单。不过还有一点需要处理，`onResourceReady()`方法和`onLoadFailed()`方法都有一个布尔值的返回值，返回`false`就表示这个事件没有被处理，还会继续向下传递，返回`true`就表示这个事件已经被处理掉了，从而不会再继续向下传递。举个简单的例子，如果我们在`RequestListener`的`onResourceReady()`方法中返回了`true`，那么就不会再回调`Target`的`onResourceReady()`方法了。

关于回调与监听的内容就讲这么多吧，如果想要学习更多深入的内容以及源码解析，还是请参考这篇文章 [Android图片加载框架最全解析（四）——玩转Glide的回调与监听](#)。

图片变换

图片变换的意思就是说，Glide从加载了原始图片到最终展示给用户之前，又进行了一些变换处理，从而能够实现一些更加丰富的图片效果，如图片圆角化、圆形化、模糊化等等。

添加图片变换的用法非常简单，我们只需要在`RequestOptions`中串接`transforms()`方法，并将想要执行的图片变换操作作为参数传入`transforms()`方法即可，如下所示：

```
RequestOptions options = new RequestOptions()
    .transforms(...);
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageView);
```

至于具体要进行什么样的图片变换操作，这个通常都是需要我们自己来写的。不过Glide已经内置了几种图片变换操作，我们可以直接拿来使用，比如`CenterCrop`、`FitCenter`、`CircleCrop`等。

但所有的内置图片变换操作其实都不需要使用`transform()`方法，Glide为了方便我们使用直接提供了现成的API：

```
RequestOptions options = new RequestOptions()
    .centerCrop();
```

```
RequestOptions options = new RequestOptions()
    .fitCenter();
```

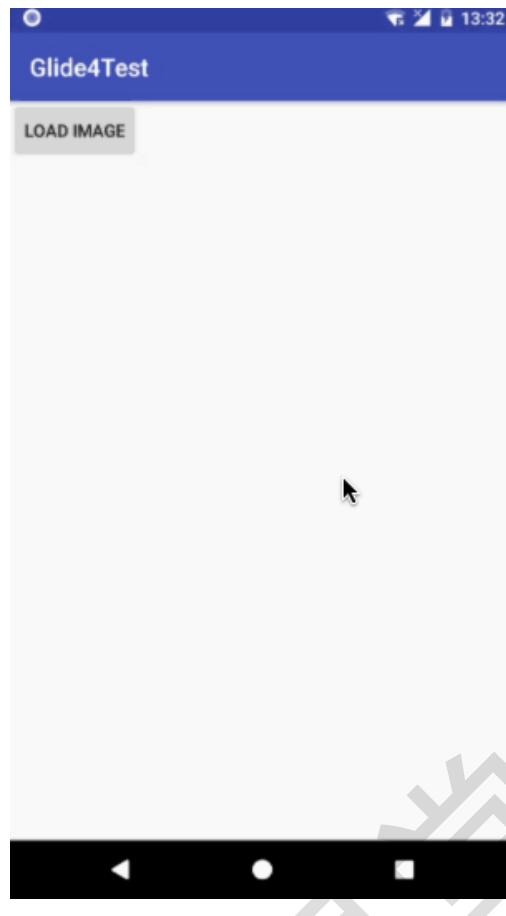
```
RequestOptions options = new RequestOptions()
    .circleCrop();
```

当然，这些内置的图片变换API其实也只是对transform()方法进行了一层封装而已，它们背后的源码仍然还是借助transform()方法来实现的。

这里我们就选择其中一种内置的图片变换操作来演示一下吧，circleCrop()方法是用来对图片进行圆形化裁剪的，我们动手试一下，代码如下所示：

```
String url = "http://guolin.tech/book.png";
RequestOptions options = new RequestOptions()
    .circleCrop();
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageView);
```

重新运行一下程序并点击加载图片按钮，效果如下图所示。



可以看到，现在展示的图片是对原图进行圆形化裁剪后得到的图片。

当然，除了使用内置的图片变换操作之外，我们完全可以自定义自己的图片变换操作。理论上，在对图片进行变换这个步骤中我们可以进行任何的操作，你想对图片怎么样都可以。包括圆角化、圆形化、黑白化、模糊化等等，甚至你将原图片完全替换成另外一张图都是可以的。

不过由于这部分内容相对于Glide 3没有任何的变化，因此就不再重复进行讲解了。想学习自定义图片变换操作的朋友们可以参考这篇文章 [Android 图片加载框架最全解析（五），Glide强大的图片变换功能](#)。

关于图片变换，最后我们再来看一个非常优秀的开源库，glide-transformations。它实现了很多通用的图片变换效果，如裁剪变换、颜色变换、模糊变换等等，使得我们可以非常轻松地进行各种各样的图片变换。

glide-transformations的项目主页地址是
<https://github.com/wasabeef/glide-transformations>。

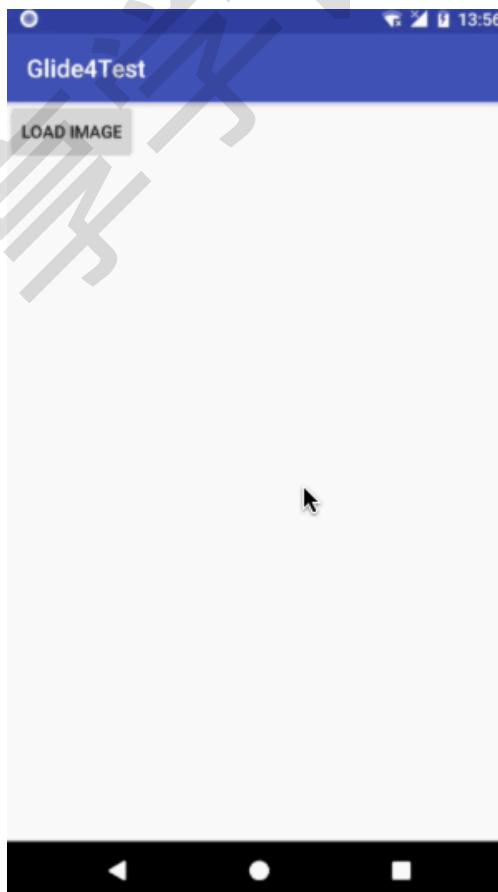
下面我们就来体验一下这个库的强大功能吧。首先需要将这个库引入到我们的项目当中，在app/build.gradle文件当中添加如下依赖：

```
dependencies {
    implementation 'jp.wasabeef:glide-
transformations:3.0.1'
}
```

我们可以对图片进行单个变换处理，也可以将多种图片变换叠加在一起使用。比如我想同时对图片进行模糊化和黑白化处理，就可以这么写：

```
String url = "http://guolin.tech/book.png";
RequestOptions options = new RequestOptions()
    .transforms(new BlurTransformation(), new
GrayscaleTransformation());
Glide.with(this)
    .load(url)
    .apply(options)
    .into(imageview);
```

可以看到，同时执行多种图片变换的时候，只需要将它们都传入到 transforms()方法中即可。现在重新运行一下程序，效果如下图所示。



当然，这只是glide-transformations库的一小部分功能而已，更多的图片变换效果你可以到它的GitHub项目主页去学习。

自定义模块

自定义模块属于Glide中的高级功能，同时也是难度比较高的一部分内容。

这里我不可能在这一篇文章中将自定义模块的内容全讲一遍，限于篇幅的限制我只能讲一讲Glide 4中变化的这部分内容。关于Glide自定义模块的全部内容，请大家去参考 [Android图片加载框架最全解析（六），探究 Glide的自定义模块功能](#) 这篇文章。

自定义模块功能可以将更改Glide配置，替换Glide组件等操作独立出来，使得我们能轻松地对Glide的各种配置进行自定义，并且又和Glide的图片加载逻辑没有任何交集，这也是一种低耦合编程方式的体现。下面我们就来学习一下自定义模块要如何实现。

首先定义一个我们自己的模块类，并让它继承自AppGlideModule，如下所示：

```
@GlideModule
public class MyAppGlideModule extends AppGlideModule {

    @Override
    public void applyOptions(Context context,
        GlideBuilder builder) {

    }

    @Override
    public void registerComponents(Context context, Glide
        glide, Registry registry) {

    }

}
```

可以看到，在MyAppGlideModule类当中，我们重写了applyOptions()和registerComponents()方法，这两个方法分别就是用来更改Glide配置以及替换Glide组件的。

注意在MyAppGlideModule类在上面，我们加入了一个@GlideModule的注解，这是Glide 4和Glide 3最大的一个不同之处。在Glide 3中，我们定义了自定义模块之后，还必须在AndroidManifest.xml文件中去注册它才能生效，而在Glide 4中是不需要的，因为@GlideModule这个注解已经能够让Glide识别到这个自定义模块了。

这样的话，我们就将Glide自定义模块的功能完成了。后面只需要在applyOptions()和registerComponents()这两个方法中加入具体的逻辑，就能实现更改Glide配置或者替换Glide组件的功能了。详情还是请参考[Android图片加载框架最全解析（六），探究Glide的自定义模块功能](#)这篇文章，这里就不再展开讨论了。

使用Generated API

Generated API是Glide 4中全新引入的一个功能，它的工作原理是使用注解处理器(Annotation Processor)来生成出一个API，在Application模块中可使用该流式API一次性调用到RequestBuilder, RequestOptions和集成库中所有的选项。

这么解释有点拗口，简单点说，就是Glide 4仍然给我们提供了一套和Glide 3一模一样的流式API接口。毕竟有些人还是觉得Glide 3的API更好用一些，比如说我。

Generated API对于熟悉Glide 3的朋友来说那是再简单不过了，基本上就是和Glide 3一模一样的用法，只不过需要把Glide关键字替换成GlideApp关键字，如下所示：

```
GlideApp.with(this)
    .load(url)
    .placeholder(R.drawable.loading)
    .error(R.drawable.error)
    .skipMemoryCache(true)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .override(Target.SIZE_ORIGINAL)
    .circleCrop()
    .into(imageView);
```

不过，有可能你的IDE中会提示找不到GlideApp这个类。这个类是通过编译时注解自动生成的，首先确保你的代码中有一个自定义的模块，并且给它加上了@GlideModule注解，也就是我们在上一节所讲的内容。然后在Android Studio中点击菜单栏Build -> Rebuild Project，GlideApp这个类就会自动生成了。

当然，Generated API所能做到的并不只是这些而已，它还可以对现有的API进行扩展，定制出任何属于你自己的API。

下面我来具体举个例子，比如说我们要求项目中所有图片的缓存策略全部都要缓存原始图片，那么每次在使用Glide加载图片的时候，都去指定diskCacheStrategy(DiskCacheStrategy.DATA)这么长长的一串代码，确实是让人比较心烦。这种情况我们就可以去定制一个自己的API了。

定制自己的API需要借助@GlideExtension和@GlideOption这两个注解。创建一个我们自定义的扩展类，代码如下所示：

```
@GlideExtension
public class MyGlideExtension {

    private MyGlideExtension() {

    }

    @GlideOption
    public static void cacheSource(RequestOptions
options) {

    options.diskCacheStrategy(DiskCacheStrategy.DATA);
}

}
```

这里我们定义了一个MyGlideExtension类，并且给加上了一个@GlideExtension注解，然后要将这个类的构造函数声明成private，这都是必须要求的写法。

接下来就可以开始自定义API了，这里我们定义了一个cacheSource()方法，表示只缓存原始图片，并给这个方法加上了@GlideOption注解。注意自定义API的方法都必须是静态方法，而且第一个参数必须是RequestOptions，后面你可以加入任意多个你想自定义的参数。

在cacheSource()方法中，我们仍然还是调用的diskCacheStrategy(DiskCacheStrategy.DATA)方法，所以说cacheSource()就是一层简化API的封装而已。

然后在Android Studio中点击菜单栏Build -> Rebuild Project，神奇的事情就会发生了，你会发现你已经可以使用这样的语句来加载图片了：

```
GlideApp.with(this)
    .load(url)
    .cacheSource()
    .into(imageView);
```

有了这个强大的功能之后，我们使用Glide就能变得更加灵活了。

结束语

这样我们基本上就将Glide 4的所有重要内容都介绍完了，如果你以前非常熟悉Glide 3的话，看完这篇文章之后相信你已经能够熟练使用Glide 4了。而如果你以前并未接触过Glide，仅仅只看这一篇文章可能了解得还不够深入，建议最好还是把前面的七篇文章也去通读一下，这样你才能成为一名Glide好手。

我翻了一下历史记录，在今年的3月21号发了这个系列的第一篇文章，用了10个月的时间终于把这个系列全部更新完了。当时承诺的是写八篇文章，如今兑现了承诺，也算是有始有终吧。未来我希望能继续给大家带来更好的技术文章，不过这个系列就到此为止了。也感谢有耐心的朋友能够看到最后，能坚持看完的人，你们都和我一样棒。

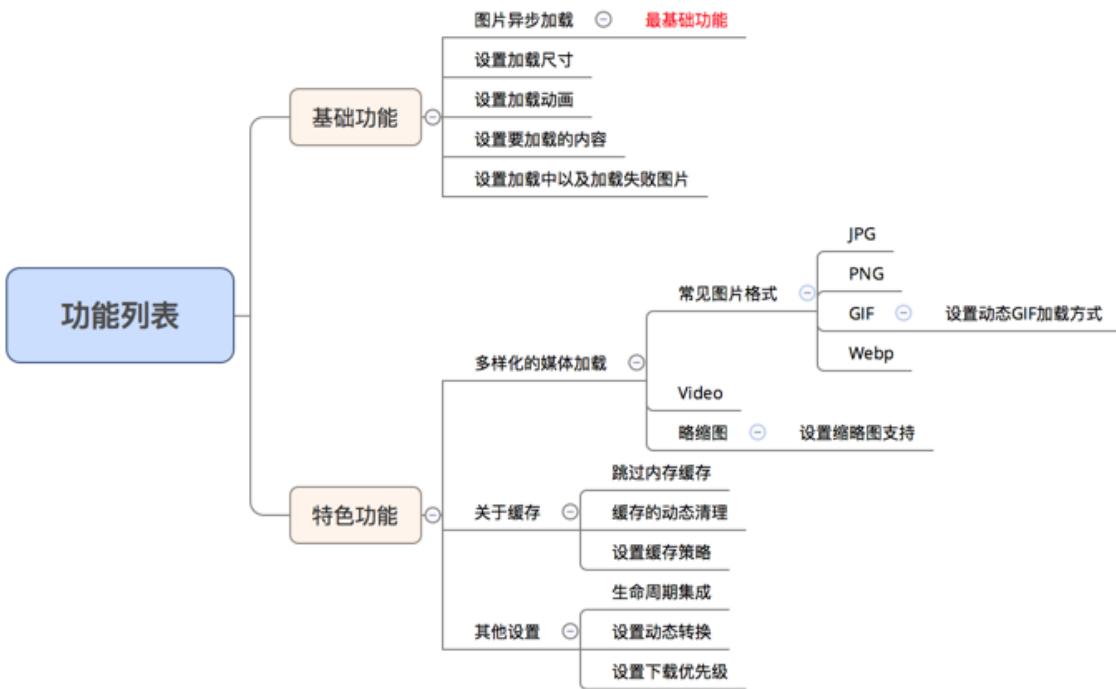
3.17 Android图片加载库：最全面解析Glide用法

1. 简介

- 介绍：Glide，是Android中一个图片加载开源库
| Google的开源项目
- 主要作用：实现图片加载
- o *

2. 功能特点

2.1 功能列表



功能列表

- 从上面可以看出，Glide不仅实现了图片异步加载的功能，还解决了Android中加载图片时需要解决的一些常见问题
- 接下来，我会对Glide的每个功能点进行详细的介绍

2.2 功能介绍

2.2.1 关于图片加载

- 图片的异步加载（基础功能）

```
ImageView targetImageView = (ImageView)
findViewById(R.id.ImageView);
String url = "http://218.192.170.132/1.jpg";
```

//Glide使用了流式接口的调用方式
//Glide类是核心实现类。

```
Glide.with(context).load(url).into(targetImageView);
```

//实现图片加载功能至少需要三个参数：
//with(Context context)
//Context对于很多Android API的调用都是必须的，这里就不多说了

```
//load(string imageUrl): 被加载图像的URL地址。
```

```
//大多情况下，一个字符串代表一个网络图片的URL。
```

```
//into(ImageView targetImageview): 图片最终要展示的地方。
```

- 设置加载尺寸

```
Glide.with(this).load(imageUrl).override(800,  
800).into(imageView);
```

- 设置加载中以及加载失败图片

api里面对placeholder()、error()函数有多态实现，用的时候可以具体的熟悉一下

```
Glide  
.with(this)  
.load(imageUrl)  
  
.placeholder(R.mipmap.ic_launcher).error(R.mipmap.ic_launcher).into(imageView);
```

- 设置加载动画

```
Glide.with(this).load(imageUrl).animate(R.anim.item_alpha  
_in).into(imageView);
```

api也提供了几个常用的动画：比如crossFade()

R.anim.item_alpha_in

```
<?xml version="1.0" encoding="utf-8"?>
<set
    xmlns:android="http://schemas.android.com/apk/res/android">
    <alpha
        android:duration="500"
        android:fromAlpha="0.0"
        android:toAlpha="1.0"/>
</set>
```

- 设置要加载的内容

项目中有很多需要先下载图片然后再做一些合成的功能，比如项目中出现的图文混排，该如何实现目标下

```
Glide.with(this).load(imageUrl).centerCrop().into(new
SimpleTarget<GlideDrawable>() {
    @Override
    public void onResourceReady(GlideDrawable
resource, GlideAnimation<? super GlideDrawable>
glideAnimation) {
        imageView.setImageDrawable(resource);
    }
});
```

2.2.2 多样式的媒体加载

```
Glide
    .with(context)
    .load(imageurl);
    .thumbnail(0.1f); //设置缩略图支持：先加载缩略图 然后在
加载全图
                                                //传了一个 0.1f 作为参数，Glide 将
会显示原始图像的10%的大小。
                                                //如果原始图像有 1000x1000 像素，那
么缩略图将会有 100x100 像素。
    .asBitmap()//显示gif静态图片
    .asGif(); //显示gif动态图片
    .into(imageview);
```

2.2.3 关于缓存

- 设置磁盘缓存策略

```
Glide.with(this).load(imageurl).diskCacheStrategy(DiskCac
heStrategy.ALL).into(imageview);

// 缓存参数说明
// DiskCacheStrategy.NONE: 不缓存任何图片，即禁用磁盘缓存
// DiskCacheStrategy.ALL : 缓存原始图片 & 转换后的图片（默认）
// DiskCacheStrategy.SOURCE: 只缓存原始图片（原来的全分辨率的图
像，即不缓存转换后的图片）
// DiskCacheStrategy.RESULT: 只缓存转换后的图片（即最终的图像：
降低分辨率后 / 或者转换后 ， 不缓存原始图片
```

- 设置跳过内存缓存

```
Glide  
    .with(this)  
    .load(imageUrl)  
    .skipMemoryCache(true)  
    .into(imageview);  
//设置跳过内存缓存  
//这意味着 Glide 将不会把这张图片放到内存缓存中去  
//这里需要明白的是，这只是会影响内存缓存！Glide 将会仍然利用磁盘缓存来避免重复的网络请求。
```

- 清理缓存

```
Glide.get(this).clearDiskCache(); //清理磁盘缓存 需要在子线程中执行  
Glide.get(this).clearMemory(); //清理内存缓存 可以在UI主线程中进行
```

2.2.4 其他设置

- 生命周期集成

通过设置绑定生命周期，我们可以更加高效的使用Glide提供的方式进行绑定，这样可以更好的让加载图片的请求的生命周期动态管理起来

```
.with(Context context) // 绑定Context  
.with(Activity activity); // 绑定Activity  
.with(FragmentActivity activity); // 绑定FragmentActivity  
.with(Fragment fragment); // 绑定Fragment
```

注意：

1. 传入的context类型影响到Glide加载图片的优化程度
2. Glide可以监视Activity的生命周期，在Activity销毁的时候自动取消等待中的请求。但是如果你使用Application context，你就失去了这种优化效果。

- 设置动态转换

```
Glide.with(this).load(imageUrl).centerCrop().into(imageview);
```

- 设置下载优先级

```
Glide.with(this).load(imageUrl).priority(Priority.NORMAL).into(imageview);
```

3. Demo实例

没有Demo的代码讲解不是好文章，让我们来一步步学会使用Glide。

步骤1：在gradle添加依赖

```
compile 'com.github.bumptech.glide:glide:3.7.0'
```

步骤2：添加网络权限

```
<uses-permission  
    android:name="android.permission.INTERNET"/>
```

步骤1和步骤2是Glide使用的前提，千万别忘了！！！

步骤3：在MainActivity中

```
ImageView targetImageView = (ImageView)
findViewById(R.id.ImageView);
String url = "http://218.192.170.132/1.jpg";

Glide
    .with(this)
    .load(url)
    .into(targetImageView);
```

| 还有具体其他功能需要配置的自己按照我上面写的进行配置就好了~

这里再贴上Glide的Github地址：[请点击这里](#)

4. 特点

4.1 优点

- 多样化媒体加载

Glide 不仅是一个图片缓存，它支持 Gif、WebP、缩略图。甚至是 Video

- 生命周期集成

通过设置绑定生命周期，我们可以更加高效的使用Glide提供的方式进行绑定，这样可以更好的让加载图片的请求的生命周期动态管理起来

- 高效的缓存策略

A. 支持Memory和Disk图片缓存

B. Picasso 只会缓存原始尺寸的图片，而 Glide 缓存的是多种规格，也就意味着 Glide 会根据你 ImageView 的大小来缓存相应大小的图片尺寸

| 比如你 ImageView 大小是200_200，原图是 400_400，而使用 Glide 就会缓存 200_200 规格的图，而 Picasso 只会缓存 400_400 规格的。这个改进就会导致 Glide 比 Picasso 加载的速度要快，毕竟少了每次裁剪重新渲染的过程，非常灵活 & 加载速度快

C. 内存开销小

默认的 Bitmap 格式是 RGB_565 格式，而 Picasso 默认的是 ARGB_8888 格式，这个内存开销要小一半。

Android关于图片内存计算，共有四种，分别是：

1. ALPHA_8：每个像素占用1byte内存
2. ARGB_4444:每个像素占用2byte内存
3. ARGB_8888:每个像素占用4byte内存（默认，色彩最细腻=显示质量最高=占用的内存也最大）
4. RGB_565:每个像素占用2byte内存（8bit = 1byte）

举例说明：一个32位的PNG=ARGB_8888=1204x1024,那么占用空间是:1024x1024x(32/8) = 4,194,304kb=4M左右

在解析图片的时候，为了避免oom和节省内存，最好使用ARGB_4444模式（节省一半的内存空间）

4.2 缺点

- 使用方法复杂

由于Glide其功能强大，所以使用的方法非常多，其源码也相对的复杂

- 包较大

- ○ *

5. 相比其他图片加载库（Picasso & Fresco）

- 对比Picasso

Glide 是在Picasso 基础之上进行的二次开发做了不少改进，不过这也导致包比 Picasso 大不少，不过也就不到 500k （Picasso 是100多 k），用法较为复杂，不过毕竟级别还是蛮小的，影响不是很大

- 对比Fresco

使用较 Fresco 简单，但性能（加载速度 & 缓存）却比不上 Fresco

- ○ *

6. 应用场景

根据Glide的特点和与其他图片加载库的对比，可以得出其使用场景：

- 需要更多的内容表现形式(如Gif)；
- 更高的性能要求（缓存 & 加载速度）；
- ○ *

7. 总结

- Glide使用起来是不是非常简单？相信你看完这篇文章后你能全面掌握 Glide的用法
- 但是Glide的源码却并不简单，接下来我会对**Glide的源码**进行详细分析。

3.18 Glide-绑定生命周期

1.with

其实Glide与activity和fragment绑定生命周期很简单,只用在with的时候传入想绑定生命周期的Context就行.

比如通常在MainActivity中传入this,或者MainActivity即可

(在Glide内部会根据你Context的实际类型做不同的处理,具体的分析会在以后的源码分析中展示)

```
Glide.with(this).load(mUrl).into(mIv);
```

3.19 Glide-内存缓存与磁盘缓存

1.缓存的资源

Glide的缓存资源分为两种:

- 1.原图(SOURCE) :原始图片
- 2.处理图(RESULT) :经过压缩和变形等处理后的图片

2.内存缓存策略(skipMemoryCache)

Glide默认是会在内存中缓存处理图(RESULT)的.

可以通过调用skipMemoryCache(true)来设置跳过内存缓存

```
//跳过内存缓存
```

```
Glide.with(this).load(murl).skipMemoryCache(true).into(mIv);
```

调用skipMemoryCache(false)没有代码上的意义,因为Glide默认就是不跳过内存缓存的,但是显示调用这个方法,可以让别人一目了然的知道你这次请求是会在内存中缓存的,所以还是建议显示调用一下这个方法来表明你的内存缓存策略

3.磁盘缓存策略(diskCacheStrategy)

Glide磁盘缓存策略分为四种,默认的是RESULT(默认值这一点网上很多文章都写错了,但是这一点很重要):

- 1.ALL:缓存原图(SOURCE)和处理图(RESULT)
- 2.NONE:什么都不缓存
- 3.SOURCE:只缓存原图(SOURCE)
- 4.RESULT:只缓存处理图(RESULT) — 默认值

4.组合策略

和其他三级缓存一样,Glide的缓存读取顺序是 内存->磁盘->网络

需要注意的是Glide的内存缓存和磁盘缓存的配置相互没有直接影响,所以可以同时进行配置

例:

1.内存不缓存,磁盘缓存缓存所有图片

```
Glide.with(this).load(murl).skipMemoryCache(true).diskCacheStrategy(DiskCacheStrategy.ALL).into(mIv);
```

2.内存缓存处理图,磁盘缓存原图

```
Glide.with(this).load(murl).skipMemoryCache(false).diskCacheStrategy(DiskCacheStrategy.SOURCE).into(mIV);
```

5.缓存大小及路径

5.1内存缓存最大空间

Glide的内存缓存其实涉及到比较多的计算,这里就介绍最重要的一个参数,就是内存缓存最大空间

内存缓存最大空间(maxSize)=每个进程可用的最大内存 * 0.4

(低配手机的话是: 每个进程可用的最大内存 * 0.33)

5.2磁盘缓存大小

磁盘缓存大小: $250 * 1024 * 1024$ (250MB)

```
/** 250 MB of cache. */
int DEFAULT_DISK_CACHE_SIZE = 250 * 1024 * 1024;
```

5.3磁盘缓存目录

磁盘缓存目录: 项目/cache/image_manager_disk_cache

```
String DEFAULT_DISK_CACHE_DIR =
"image_manager_disk_cache";
```

6.清除缓存

6.1清除所有缓存

清除所有内存缓存(需要在Ui线程操作)

```
Glide.get(this).clearMemory();
```

清除所有磁盘缓存(需要在子线程操作)

```
Glide.get(MainActivity.this).clearDiskCache();
```

注:在使用中的资源不会被清除

6.2 清除单个缓存

由于Glide可能会缓存一张图片的多个分辨率的图片,并且文件名是被哈希过的,所以并不能很好的删除单个资源的缓存,以下是官方文档中的描述

Because File names are hashed keys, there is no good way to simply delete all of the cached files on disk that correspond to a particular url or file path. The problem would be simpler if you were only ever allowed to load or cache the original image, but since Glide also caches thumbnails and provides various transformations, each of which will result in a new File in the cache, tracking down and deleting every cached version of an image is difficult.

In practice, the best way to invalidate a cache file is to change your identifier when the content changes (url, uri, file path etc).

3.20 Glide-通过Modules定制Glide

前言:

我们一般情况下使用Glide都很简单,只用简单的调用几个方法就能够很好的显示图片了,但其实Glide在初始化的时候进行了一系列的默认配置,比如缓存的配置,图片质量的配置等等.接下来我们就介绍一下一个比较高级的功能,通过Modules定制自己的个性Glide

1. 创建一个类实现GlideModule

```
public class xiayuGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        //TODO
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
        //TODO
    }
}
```

2.配置清单文件

在AndroidManifest中配置刚刚创建的GlideModule,需要在application节点下添加

```
<application>
    ...
    <meta-data
        android:name="com.xiayu.xiayuglidedemo.xiayuGlideModule"
        android:value="GlideModule" />
</application>
```

其中android:name就是刚才创建的GlideModule的实现类

3.进行自定义配置

刚才创建的GlideModule的实现类时,会要实现两个方法,这里要用到的是其中的applyOptions方法,applyOptions方法里面提供了一个GlideBuilder,通过GlideBuilder我们就能实现自定义配置了

```
public class XiayuGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {

        builder.setDiskCache(); //自定义磁盘缓存

        builder.setMemoryCache(); //自定义内存缓存

        builder.setBitmapPool(); //自定义图片池

        builder.setDiskCacheService(); //自定义本地缓存的线程
池

        builder.setResizeService(); //自定义核心处理的线程池

        builder.setDecodeFormat(); //自定义图片质量

    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
        //TO
    }
}
```

4.例子(配置默认图片质量)

由于Glide的配置涉及到的东西比较多,在以后的文章里面会对每一种配置做说明具体介绍,这里就先示范一个比较简单的配置,那就是图片质量配置

用过Picasso的朋友应该知道,Picasso默认的图片质量是ARGB_8888,而Glide默认的图片质量是RGB_565,这里我们就来修改默认配置,让Glide的默认质量为ARGB_8888

(ARGB_8888是指32位图,即每个像素占4byte)

(RGB_565是16位图,即每个像素占2byte)

```
public class XiayuGlideModule implements GlideModule {  
  
    @Override  
    public void applyOptions(Context context,  
    GlideBuilder builder) {  
  
        //自定义图片质量  
  
        builder.setDecodeFormat(DecodeFormat.PREFER_ARGB_8888);  
  
    }  
  
    @Override  
    public void registerComponents(Context context, Glide  
    glide) {  
        //TO  
    }  
}
```

5.其他

在清单文件中可以配置多个GlideModule,Glide会依次遍历并读取

3.21 Glide-自定义缓存

前言:

在前面的文章中,我们介绍了如何对Glide进行默认配置的基础知识,现在我们就来介绍如何配置自定义缓存

1.如何使用GlideModule

首先我们要对Glide进行默认配置,需要了解如何使用GlideModule,前面文章已经做了详细介绍了,这边就不再介绍了

如何使用GlideModule

<http://blog.csdn.net/yulyu/article/details/55194237>

前面还介绍了关于Glide缓存的基础知识,有兴趣的也可以看看

Glide缓存基础知识

<http://blog.csdn.net/yulyu/article/details/55096713>

2.自定义内存缓存

通过调用builder的setXXX方法就可以自定义内存缓存的大小了

(MemoryCache以及BitmapPool都与内存缓存有关)

```
public class XiayuGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        //配置内存缓存大小 10MB
        builder.setMemoryCache(new
LruResourceCache(10*1024*1024));
        //配置图片池大小 20MB
        builder.setBitmapPool(new
LruBitmapPool(20*1024*1024));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {
    }
}
```

但是内存缓存的大小数值其实不应该是随便配置的,Glide的内部的默认值是通过一系列的计算获取的,比如说判断手机是否高配置手机等(有兴趣的可以去看他的源码,或者等待本系列后面的源码分析)
这样有个取巧的办法,就是获取Glide默认的设置,然后在这个设置的基础上进行修改

```
public class XiayuGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        //获取内存计算器
        MemorySizeCalculator calculator = new
        MemorySizeCalculator(context);
        //获取Glide默认内存缓存大小
        int defaultMemoryCacheSize =
        calculator.getMemoryCacheSize();
        //获取Glide默认图片池大小
        int defaultBitmapPoolSize =
        calculator.getBitmapPoolSize();
        //将数值修改为之前的1.1倍
        int myMemoryCacheSize = (int)
        (1.1 * defaultMemoryCacheSize);
        int myBitmapPoolSize = (int)
        (1.1 * defaultBitmapPoolSize);
        //修改默认值
        builder.setMemoryCache(new
        LruResourceCache(myMemoryCacheSize));
        builder.setBitmapPool(new
        LruBitmapPool(myBitmapPoolSize));
    }

    @Override
    public void registerComponents(Context context, Glide
    glide) {

}
```

```
}
```

3.自定义磁盘缓存

磁盘缓存按照访问权限及路径可以分为两种

- 私有缓存(自己本app可以使用,目录在data/data/应用包名下面)
- 外部缓存(都可以访问,目录在扩展空间内,如SD卡)

3.1私有缓存

下面这样配置的话,主要是修改缓存大小,缓存路径仍为默认路径(即
data/data/应用包名/cache/image_manager_disk_cache)

```
public class XiayuGlideModule implements GlideModule {

    @Override
    public void applyOptions(Context context,
    GlideBuilder builder) {
        //设置磁盘缓存大小
        int size = 100 * 1024 * 1024;
        //设置磁盘缓存
        builder.setDiskCache(new
InternalCacheDiskCacheFactory(context, size));
    }

    @Override
    public void registerComponents(Context context, Glide
glide) {

    }
}
```

如果需求修改缓存路径的话,需要增加一下参数即可(变为data/data/应用
包名/cache/xiayu)

```
//设置磁盘缓存大小  
int size = 100 * 1024 * 1024;  
String dir = "xiayu";  
//设置磁盘缓存  
builder.setDiskCache(new  
InternalCacheDiskCacheFactory(context, dir, size));
```

3.2外部缓存

下面这样配置的话,缓存会在 SDCard/Android/data/应用包名/cache/image_manager_disk_cache目录下

```
public class XiayuGlideModule implements GlideModule {  
    @Override public void applyOptions(Context context,  
        GlideBuilder builder) { //设置磁盘缓存大小 int  
        size = 100 * 1024 * 1024; //设置磁盘缓存  
        builder.setDiskCache(new  
            ExternalCacheDiskCacheFactory(context, size)); }  
    @Override public void registerComponents(Context  
        context, Glide glide) { }}
```

如果需求修改缓存路径的话,需要增加一下参数即可(变为 SDCard/Android/data/应用包名/cache/xiayu)

```
//设置磁盘缓存大小 int size = 100 * 1024 * 1024;  
String dir = "xiayu"; //设置磁盘缓存  
builder.setDiskCache(new  
ExternalCacheDiskCacheFactory(context, dir, size));
```

3.3完全自定义路径

上面两种缓存都是把缓存的父目录定死的,能够改变的只是子目录,Glide还提供了一种方式可以完全自定义缓存目录

(需要注意的是这里的路径是配置的绝对路径,所以如果没有指定在sd卡目录下的话是无法直接看到的)

```
public class XiayuGlideModule implements GlideModule {  
    @Override public void applyOptions(Context context,  
        GlideBuilder builder) { //设置磁盘缓存大小 int  
        size = 100 * 1024 * 1024; String dirFolder =  
        "xia"; String dirName = "yu"; //设置磁盘缓存  
        builder.setDiskCache(new  
        DiskLruCacheFactory(dirFolder, size));  
        //builder.setDiskCache(new DiskLruCacheFactory(dirFolder,  
        dirName, size)); } @Override public void  
    registerComponents(Context context, Glide glide) { }}
```

3.22 Glide-图片的压缩

前言:

这一节里面我们将介绍Glide如何对图片进行压缩,这一点在加载图片较多或者加载的图片像素很高的程序里面至关重要

1.Android图片显示相关知识

这里会讲一下图片显示相关的基础知识,如果不关心的可以直接跳到第二点,不过建议是最好看一下

1.1图片质量分类

安卓图片显示的质量配置主要分为四种:

- ARGB_8888 :32位图,带透明度,每个像素占4个字节
- ARGB_4444 :16位图,带透明度,每个像素占2个字节
- RGB_565 :16位图,不带透明度,每个像素占2个字节
- ALPHA_8 :32位图,只有透明度,不带颜色,每个像素占4个字节

(A代表透明度,RGB代表红绿蓝:即颜色)

1.2图片默认质量

Picasso的默认质量是 ARGB_8888

Glide的默认质量则为 RGB_565

1.3占用内存

加载一张4000 * 2000(一般手机拍摄的都超过这个像素)的图片

Picasso需要占用的内存为: 32MB

$$4000 * 2000 * 4 / 1024 / 1024 = 30 \text{ (MB)}$$

Glide需要占用的内存为: 16MB

$$4000 * 2000 * 2 / 1024 / 1024 = 15 \text{ (MB)}$$

也就是说只要同时加载几张图片,你的应用就会OOM(内存溢出了),最恐怖的是就算你的ImageView的宽高只有10px,同样会占用那么多内存,这就是为什么需要做图片压缩的原因了

2.图片质量的压缩或者提高

在Glide里面默认用的是RGB_565,如果需要修改,相对于Picasso来说会更加麻烦一些,在之前的文章中,讲到过如何修改Glide的图片质量,这里就不再做介绍了

[通过GlideModule定制Glide](#)

<http://blog.csdn.net/yulyu/article/details/55194237>

3.图片尺寸的压缩或者拉伸 override)

通过调用override,就可以把图片压缩到相应的尺寸来显示了,类似这些被处理过的图片,就是之前文章中提到的RESULT(处理图)

```
Glide.with(this).load(murl).override(300,300).into(mIV);
```

注意,这里具体会压缩到什么尺寸还会根据很多条件来计算,所以最终压缩的结果的宽高会比较接近你的传值,但是不一定会完全相同,如果感兴趣的可以期待本系列后面的Glide源码解析中的具体介绍

(一般来说控件是什么尺寸就传入相应的宽高,这样是比较适合的压缩比例)

3.23 Glide-图片预处理(圆角,高斯模糊等)

前言:

之前已经讲解过如何简单的显示图片,但是有时候项目中会有很多特殊的需求,比如说圆角处理,圆形图片,高斯模糊等,Glide提供了方法可以很好的进行处理,接下来我们就介绍一下

1.创建一个类继承BitmapTransformation

需要实现两个方法,其中transform方法里面能拿到bitmap对象,这里就是对图片做处理的地方

```
public class CornersTransform extends  
BitmapTransformation { @Override protected  
Bitmap transform(BitmapPool pool, Bitmap toTransform, int  
outWidth, int outHeight) { //TODO }  
@Override public String getId() {  
//TODO } }
```

2.使用

通过调用transform方法就能展示处理后的图片

```
Glide.with(this).load(url).transform(new  
CornersTransform()).into(iv1);
```

3.举例(圆角处理)

3.1 自定义Transformation

```
public class CornersTransform extends  
BitmapTransformation { private float radius; public  
CornersTransform(Context context) {  
super(context); radius = 10; } public  
CornersTransform(Context context, float radius) {  
super(context); this.radius = radius; }  
@Override protected Bitmap transform(BitmapPool pool,  
Bitmap toTransform, int outWidth, int outHeight) {  
return cornersCrop(pool, toTransform); } private  
Bitmap cornersCrop(BitmapPool pool, Bitmap source) {  
if (source == null) return null; Bitmap result  
= pool.get(source.getWidth(), source.getHeight(),  
Bitmap.Config.ARGB_8888); if (result == null) {  
result = Bitmap.createBitmap(source.getWidth(),  
source.getHeight(), Bitmap.Config.ARGB_8888); }  
Canvas canvas = new Canvas(result); Paint  
paint = new Paint(); paint.setShader(new  
BitmapShader(source, BitmapShader.TileMode.CLAMP,  
BitmapShader.TileMode.CLAMP));  
paint.setAntiAlias(true); RectF rectF = new  
RectF(0f, 0f, source.getWidth(), source.getHeight());  
canvas.drawRoundRect(rectF, radius, radius, paint);  
return result; } @Override public String  
getId() { return getClass().getName(); }}
```

3.2 使用

```
Glide.with(this).load(url).transform(new  
CornersTransform(this,50)).into(iv1);
```

效果如下：



4. 使用多个transform

transform方法是不支持多次调用的,如果你调用了两次,那么第二次的会覆盖了第一次的效果

但是他有一个重载的方法可以传入多个对象,这样传入的变形器都能够生效

```
Glide.with(this).load(url).transform(new  
CircleTransform(this),new  
CornersTransform(this,50)).into(iv1);
```

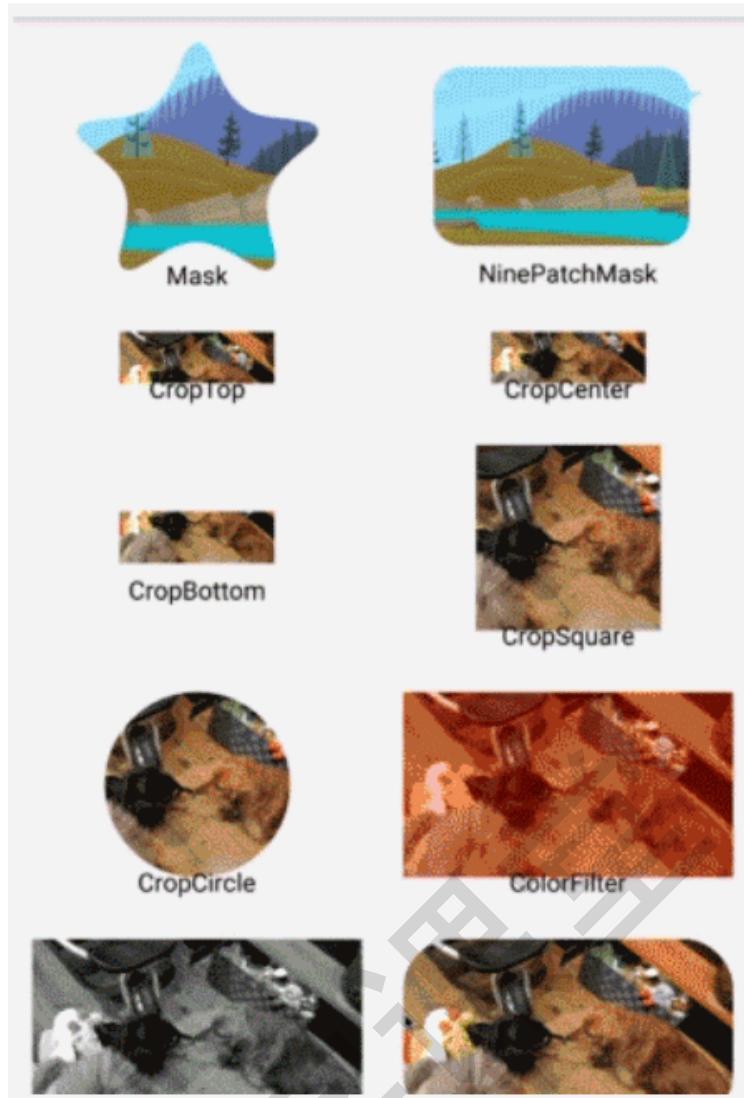
5.三方库

如果你觉得自己自定义transform比较困难,或者你想学习别人的图片处理方法,可以在试一试github上的这个三方库

[Glide Transformations](#)

<https://github.com/wasabeef/glide-transformations>

效果(支持圆角,高斯模糊等)



3.24 Glide-图片的剪裁(ScaleType)

前言：

这一节里面我们将讲到关于Glide的图片的剪裁

1.基础知识

这里会涉及到Glide的变换处理transform,在之前的文章中已经讲过了,这里就不做介绍了,没有看过的朋友最好去了解一下,不然接下来的分析可能会理解不了

2.ImageView默认的ScaleType

讲到图片的剪裁,我们首先要介绍一下ImageView默认的ScaleType设置效果

ImageView的ScaleType一共有8种属性:

- matrix
- center
- centerInside
- centerCrop
- fitCenter(默认)
- fitStart
- fitEnd
- fitXY

有些文章说默认是matrix,是不正确的,其实默认是FIT_CENTER

可以通过ImageView的源码看到默认设置

```
private void initImageView() {    mMatrix = new Matrix();  
    mScaleType = ScaleType.FIT_CENTER;    if  
(!sCompatDone) {        final int targetSdkVersion =  
mContext.getApplicationInfo().targetSdkVersion;  
sCompatAdjustViewBounds = targetSdkVersion <=  
Build.VERSION_CODES.JELLY_BEAN_MR1;  
sCompatUseCorrectStreamDensity = targetSdkVersion >  
Build.VERSION_CODES.M;  
sCompatDrawableVisibilityDispatch = targetSdkVersion <  
Build.VERSION_CODES.N;            sCompatDone = true;    }}
```

2.1属性介绍

(1)matrix

不缩放,图片与控件 **左上角** 对齐,当图片大小超过控件时将被 **裁剪**

(2)center

不缩放,图片与控件 **中心点** 对齐,当图片大小超过控件时将被 **裁剪**

(3)centerInside

以完整显示图片为目标, 不剪裁,当显示不下的时候将缩放,能够显示的情况下不缩放**

(4)centerCrop

以填满整个控件为目标,等比缩放,超过控件时将被 **裁剪** (**宽高都要填满**,所以只要图片宽高比与控件宽高比不同时,一定会被剪裁)

(5)fitCenter(默认)

自适应控件, 不剪裁,在不超过控件的前提下,等比 **缩放** 到 **最大**,**居中**显示

(6)fitStart

自适应控件, 不剪裁,在不超过控件的前提下,等比 **缩放** 到 **最大**,**靠左(上)**显示

(7)fitEnd

自适应控件, 不剪裁,在不超过控件的前提下,等比 **缩放** 到 **最大**,**靠右(下)**显示

(8)fitXY

以填满整个控件为目标, 不按比例 拉伸或缩放(**可能会变形**), **不剪裁**

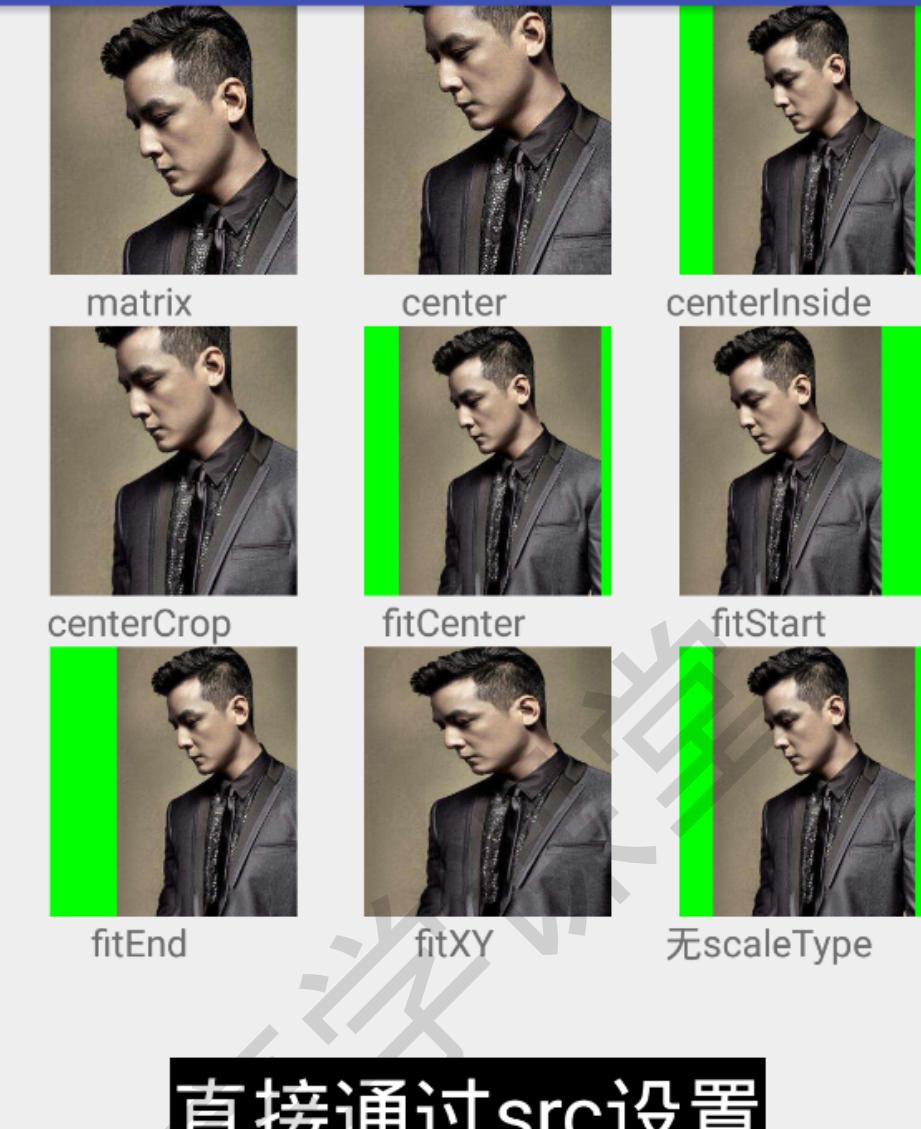
2.2效果图

当你不使用Glide,直接在ImageView的xml的src中设置图片效果如下

图片 336 * 448

控件 300 * 300

XiayuGlideDemo



3.Glide配置

Glide有两个方法可以设置图片剪裁的策略

①fitCenter()

②centerCrop()

这两个方法其实都是通过调用transform方法来对图片进行处理

3.1默认策略

当你没有调用上述两个方法,并且也没有调用transform方法的时候,在Glide调用into方法时,会根据你设置的ScaleType来做处理

```
if (!isTransformationSet && view.getScaleType() !=  
null) { switch (view.getScaleType()) {  
case CENTER_CROP: applyCenterCrop();  
break;  
case FIT_START: case FIT_CENTER:  
applyFitCenter(); case FIT_END:  
break;  
//$CASES-OMITTED$ default: //  
Do nothing. } }
```

applyCenterCrop其实还是调用的centerCrop

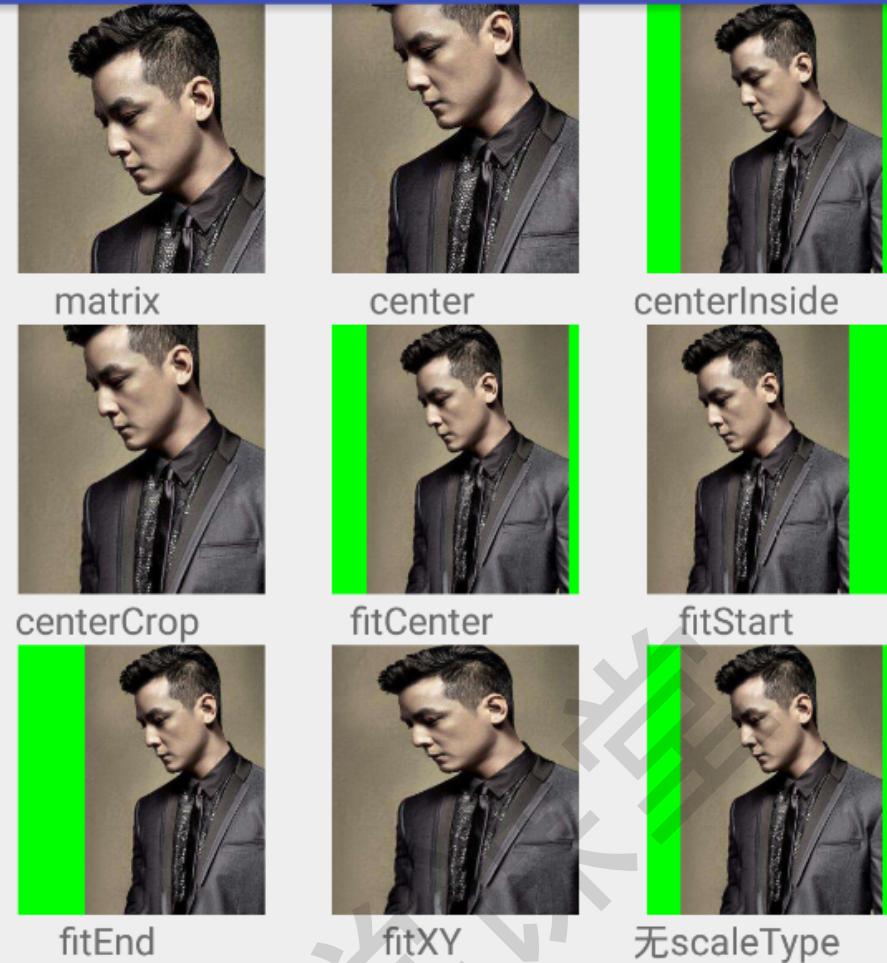
applyFitCenter其实还是调用的applyFitCenter

通过默认的配置加载图片

```
Glide.with(this).load(url).into(iv1);
```

效果如下(跟ImageView设置src一样)

XiayuGlideDemo



Glide默认的配置

<http://blog.csdn.net/yulyu>

3.2 fitCenter

内部是调用transform方法来处理的,处理代码这边只是展示一下,就不做详细的介绍了,有兴趣的朋友可以研究

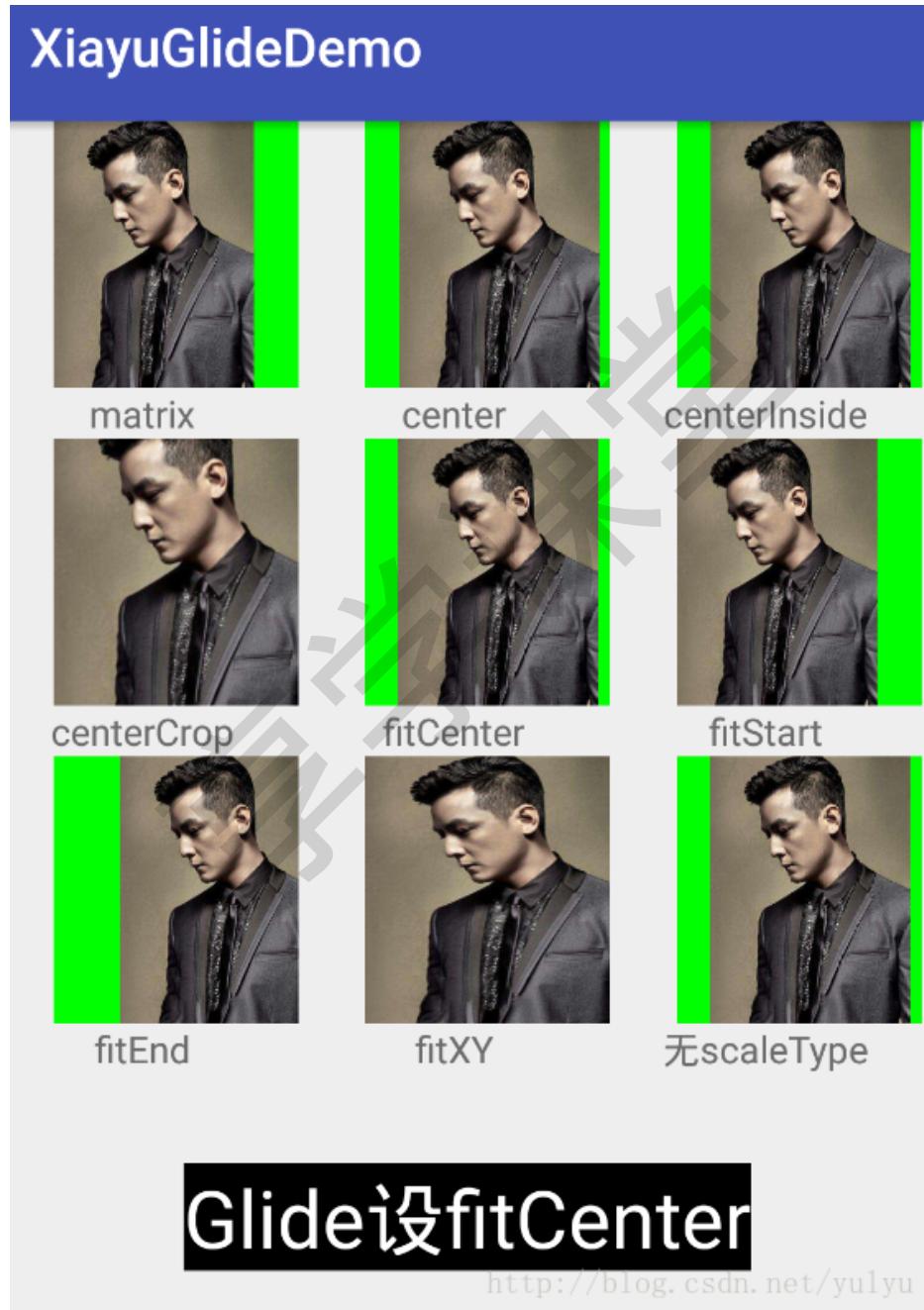
想
識
思
學

```
public static Bitmap fitCenter(Bitmap toFit, BitmapPool
pool, int width, int height) {           if
(toFit.getWidth() == width && toFit.getHeight() ==
height) {           if (Log.isLoggable(TAG,
Log.VERBOSE)) {                   Log.v(TAG, "requested
target size matches input, returning input");
}           return toFit;       }       final float
widthPercentage = width / (float) toFit.getWidth();
final float heightPercentage = height / (float)
toFit.getHeight();       final float minPercentage =
Math.min(widthPercentage, heightPercentage);       //
take the floor of the target width/height, not round. If
the matrix       // passed into drawBitmap rounds
differently, we want to slightly       // overdraw, not
underdraw, to avoid artifacts from bitmap reuse.
final int targetWidth = (int) (minPercentage *
toFit.getWidth());       final int targetHeight = (int)
(minPercentage * toFit.getHeight());       if
(toFit.getWidth() == targetWidth && toFit.getHeight() ==
targetHeight) {           if (Log.isLoggable(TAG,
Log.VERBOSE)) {                   Log.v(TAG, "adjusted
target size matches input, returning input");
}           return toFit;       }       Bitmap.Config
config = getSafeConfig(toFit);       Bitmap toReuse =
pool.get(targetWidth, targetHeight, config);       if
(toReuse == null) {           toReuse =
Bitmap.createBitmap(targetWidth, targetHeight, config);
}       // We don't add or remove alpha, so keep
the alpha setting of the Bitmap we were given.
TransformationUtils.setAlpha(toFit, toReuse);       if
(Log.isLoggable(TAG, Log.VERBOSE)) {
Log.v(TAG, "request: " + width + "x" + height);
Log.v(TAG, "toFit: " + toFit.getWidth() + "x" +
toFit.getHeight());           Log.v(TAG, "toReuse: " +
toReuse.getWidth() + "x" + toReuse.getHeight());
Log.v(TAG, "minPct: " + minPercentage);       }
Canvas canvas = new Canvas(toReuse);       Matrix
matrix = new Matrix();
```

```
matrix.setScale(minPercentage, minPercentage);
Paint paint = new Paint(PAINT_FLAGS);
canvas.drawBitmap(toFit, matrix, paint);           return
toReuse;    }
```

效果图：

```
Glide.with(this).load(url).fitCenter().into(iv1);
```



3.3 centerCrop

跟fitCenter的原理一样,只是具体处理逻辑不一样

```
public static Bitmap centerCrop(Bitmap recycled, Bitmap toCrop, int width, int height) {    if (toCrop == null) {        return null;    } else if (toCrop.getWidth() == width && toCrop.getHeight() == height) {        return toCrop;    } // From ImageView/Bitmap.createScaledBitmap.    final float scale;    float dx = 0, dy = 0;    Matrix m = new Matrix();    if (toCrop.getWidth() * height > width * toCrop.getHeight()) {        scale = (float) height / (float) toCrop.getHeight();        dx = (width - toCrop.getWidth() * scale) * 0.5f;    } else {        scale = (float) width / (float) toCrop.getWidth();        dy = (height - toCrop.getHeight() * scale) * 0.5f;    }    m.setScale(scale, scale);    m.postTranslate((int) (dx + 0.5f), (int) (dy + 0.5f));    final Bitmap result;    if (recycled != null) {        result = recycled;    } else {        result = Bitmap.createBitmap(width, height, getSafeConfig(toCrop));    } // We don't add or remove alpha, so keep the alpha setting of the Bitmap we were given.    TransformationUtils.setAlpha(toCrop, result);    Canvas canvas = new Canvas(result);    Paint paint = new Paint(PAINT_FLAGS);    canvas.drawBitmap(toCrop, m, paint);    return result;}
```

效果图(所有的都是与centerCrop效果一样):

```
Glide.with(this).load(url).centerCrop().into(iv2);
```

XiayuGlideDemo



matrix



center



centerInside



centerCrop



fitCenter



fitStart



fitEnd



fitXY



无scaleType

Glide设centerCrop

<http://blog.csdn.net/yulyu>

4.统一展示(方便对比)

XiayuGlideDemo



matrix



center



centerInside



centerCrop



fitCenter



fitStart



fitEnd



fitXY



无scaleType

直接通过src设置

<http://blog.csdn.net/yulyu>

XiayuGlideDemo



matrix



center



centerInside



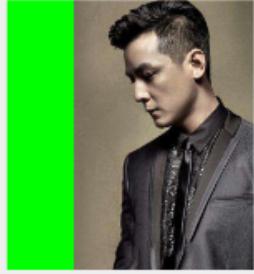
centerCrop



fitCenter



fitStart



fitEnd



fitXY



无scaleType

Glide默认的配置

<http://blog.csdn.net/yulyu>

XiayuGlideDemo



matrix



center



centerInside



centerCrop



fitCenter



fitStart



fitEnd



fitXY

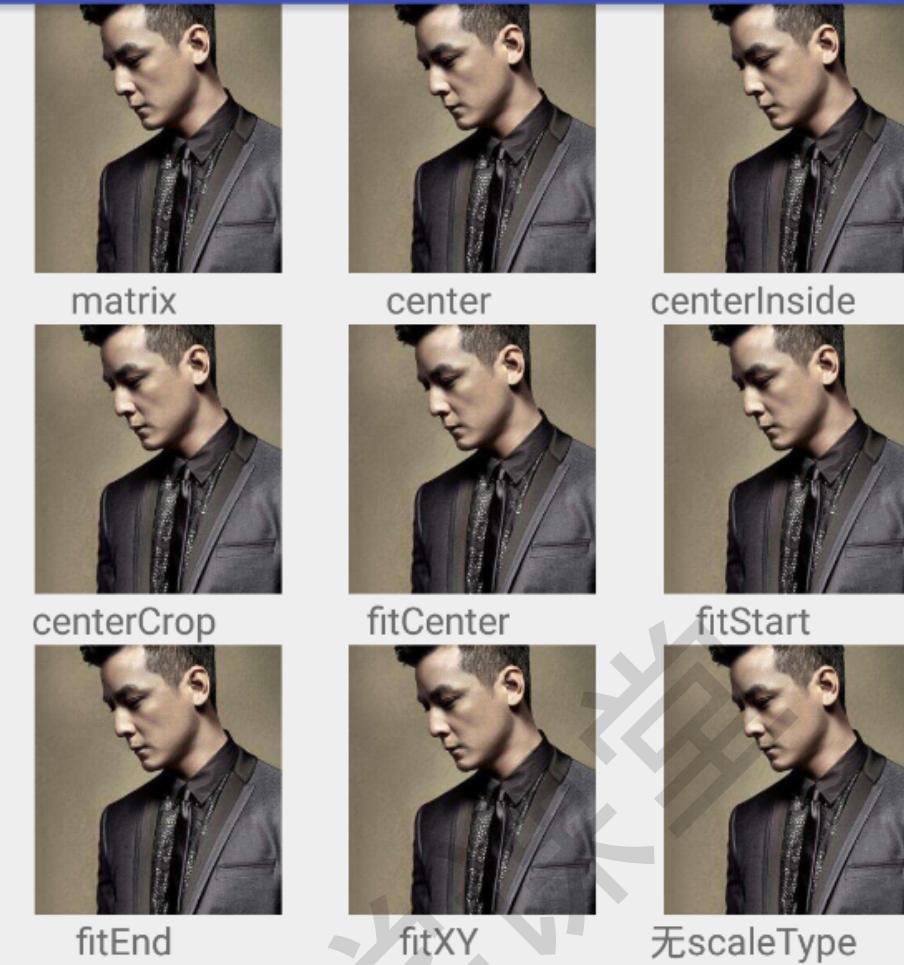


无scaleType

Glide设fitCenter

<http://blog.csdn.net/yulyu>

XiayuGlideDemo



Glide设centerCrop

<http://blog.csdn.net/yulyu>

5.其他

有一点要注意的就是fitCenter和centerCrop方法与transform方法可以共存,但是有时候会互相影响,如果说圆角处理遇到了剪裁,圆角那一部分可能会刚好被剪裁掉了

这一章节涉及到的主要是取图方面的知识,如果是对取图方面要求不高的项目,那么用原生的scaleType或者是Glide提供的两个方法即可,但是如果是取图规则特别复杂的项目(比如我现在的项目(〒_〒)),那么就需要通过自定义transform了,具体可以参考Glide的两个transform的处理,也可以用github上不错的三方库

3.25 Glide-源码详解

前言：

之前的文章中,笔者介绍了很多Glide的使用方法,但是由于Glide框架封装得太好了,很多人在使用的时候,只是知其然不知其所以然,为了不要仅仅成为“cv工程师”,只会复制粘贴,所以这篇文章我们就一起来研究一下Glide的源码,看看Glide到底是怎么将一张图片加载出来的~

前方高能预警,本文篇幅较长,阅读需要耐心

本文基于Glide 3.7.0版本

一.Glide的构造

```
//Glide.java Glide(Engine engine, MemoryCache  
memoryCache, BitmapPool bitmapPool, Context context,  
DecodeFormat decodeFormat) { ... }
```

Glide是通过GlideBuilder中的createGlide方法生成的(核心代码如下)

```
//GlideBuilder.java Glide createGlide() { ...  
return new Glide(engine, memoryCache, bitmapPool,  
context, decodeFormat); }
```

Glide的构造参数主要有四个,都是通过createGlide生成的.

- MemoryCache 内存缓存
- BitmapPool 图片池
- DecodeFormat 图片格式
- Engine 引擎类

1.MemoryCache :内存缓存 LruResourceCache

```
//MemorySizecalculator.java
final int maxSize = getMaxSize(activityManager);
private static int getMaxSize(ActivityManager activityManager) { //每个进程可用的最大内存
    final int memoryClassBytes = activityManager.getMemoryClass() * 1024 * 1024; //判断是否低配手机
    final boolean isLowMemoryDevice = isLowMemoryDevice(activityManager);
    return Math.round(memoryClassBytes * (isLowMemoryDevice ? LOW_MEMORY_MAX_SIZE_MULTIPLIER : MAX_SIZE_MULTIPLIER));
}
```

最大内存:如果是低配手机,就每个进程可用的最大内存乘以0.33,否则就每个进程可用的最大内存乘以0.4

```
//MemorySizecalculator.java
int screenSize = screenDimensions.getWidthPixels() *
screenDimensions.getHeightPixels() * BYTES_PER_ARGB_8888_PIXEL; (宽*高*4)
int targetPoolSize = screenSize * BITMAP_POOL_TARGETSCREENS; (宽*高*4*4)
int targetMemoryCacheSize = screenSize * MEMORY_CACHE_TARGETSCREENS; (宽*高*4*2)
//判断是否超过最大值,否则就等比缩小
if (targetMemoryCacheSize + targetPoolSize <= maxSize) {
    memoryCacheSize = targetMemoryCacheSize;
    bitmapPoolSize = targetPoolSize;
} else {
    int part = Math.round((float) maxSize / (BITMAP_POOL_TARGETSCREENS + MEMORY_CACHE_TARGETSCREENS));
    memoryCacheSize = part * MEMORY_CACHE_TARGETSCREENS;
    bitmapPoolSize = part * BITMAP_POOL_TARGETSCREENS;
}
```

targetPoolSize 和 targetMemoryCacheSize 之和不能超过maxSize 否则就等比缩小

```
//GlideBuilder.java
memoryCache = new LruResourceCache(calculator.getMemoryCacheSize());
```

内存缓存用的是targetMemoryCacheSize (即一般是缓存大小是屏幕的宽 * 高 * 4 * 2)

2.BitmapPool 图片池 LruBitmapPool

```
int size = calculator.getBitmapPoolSize(); bitmapPool =  
new LruBitmapPool(size);
```

图片池用的是targetPoolSize(即一般是缓存大小是屏幕的宽*高*4*4)

3.DecodeFormat 图片格式

```
DecodeFormat DEFAULT = PREFER_RGB_565
```

默认是RGB_565

4.Engine 引擎类

```
//GlideBuilder.java engine = new Engine(memoryCache,  
diskCacheFactory, diskCacheService, sourceService);
```

engine 里面主要参数

- 内存缓存 memoryCache
- 本地缓存 diskCacheFactory
- 处理源资源的线程池 sourceService
- 处理本地缓存的线程池 diskCacheService

(1)memoryCache:内存缓存 LruBitmapPool

上面已经做了介绍

(2)diskCacheFactory:本地缓存 DiskLruCacheFactory

```
//DiskCache.java/** 250 MB of cache. */ int  
DEFAULT_DISK_CACHE_SIZE = 250 * 1024 * 1024;  
String DEFAULT_DISK_CACHE_DIR =  
"image_manager_disk_cache";
```

默认大小:250 MB

默认目录:image_manager_disk_cache

(3)sourceService 处理源资源的线程池 (ThreadPoolExecutor的子类)

```
final int cores = Math.max(1,  
Runtime.getRuntime().availableProcessors());//获得可用的处理器个数  
sourceService = new  
FifoPriorityThreadPoolExecutor(cores);
```

线程池的核心线程数量等于获得可用的处理器个数

(4)diskCacheService 处理本地缓存的线程池 (ThreadPoolExecutor的子类)

```
diskCacheService = new FifoPriorityThreadPoolExecutor(1);
```

线程池的核心线程数量为1

二.with方法

with方法有很多重载,最后会返回一个RequestManager

```
//Glide.java/** * @see #with(android.app.Activity) * @see  
#with(android.app.Fragment) * @see  
#with(android.support.v4.app.Fragment) * @see  
#with(android.support.v4.app.FragmentActivity) * * @param  
context Any context, will not be retained. * @return A  
RequestManager for the top level application that can be  
used to start a load. */public static RequestManager  
with(Context context) { RequestManagerRetriever  
retriever = RequestManagerRetriever.get(); return  
retriever.get(context);}
```

就算你传入的是Context ,这里也会根据你Context 实际的类型,走不同的分支

```
//RequestManagerRetriever.java
public RequestManager
get(Context context) {    if (context == null) {
throw new IllegalArgumentException("You cannot start a
load on a null Context");    } else if
(util.isOnMainThread() && !(context instanceof
Application)) {        if (context instanceof
FragmentActivity) {            return
get((FragmentActivity) context);        } else if
(context instanceof Activity) {            return
get((Activity) context);        } else if (context
instanceof ContextWrapper) {            return
get(((ContextWrapper) context).getBaseContext());
}    }    return getApplicationManager(context);}
```

这里以FragmentActivity为例,最后会创建一个无界面的Fragment,即SupportRequestManagerFragment,让请求和你的activity的生命周期同步

```
//RequestManagerRetriever.java
public RequestManager
get(FragmentActivity activity) {    if
(util.isOnBackgroundThread()) {        return
get(activity.getApplicationContext());    } else {
assertNotDestroyed(activity);        FragmentManager fm =
activity.getSupportFragmentManager();        return
supportFragmentGet(activity, fm);    }}RequestManager
supportFragmentGet(Context context, FragmentManager fm) {
    SupportRequestManagerFragment current =
getSupportRequestManagerFragment(fm);    RequestManager
requestManager = current.getRequestManager();    if
(requestManager == null) {        requestManager = new
RequestManager(context, current.getLifecycle(),
current.getRequestManagerTreeNode());
current.setRequestManager(requestManager);    }    return
requestManager;}
```

这里需要注意一下,如果你是在子线程调用with方法,或者传入的Context是Application的话,请求是跟你的Application的生命周期同步

```
//RequestManagerRetriever.java
private RequestManager
getApplicationManager(Context context) { // Either an
application context or we're on a background thread.
if (applicationManager == null) { synchronized
(this) { if (applicationManager == null) {
// Normally pause/resume is taken care of by
the fragment we add to the fragment or activity.
// However, in this case since the manager
attached to the application will not receive lifecycle
// events, we must force the manager to
start resumed using ApplicationLifecycle.
applicationManager = new
RequestManager(context.getApplicationContext(),
new ApplicationLifecycle(), new
EmptyRequestManagerTreeNode()); } } }
} return applicationManager; }
```

三.load方法

这里方法也有很多重载

```
//RequestManager.java
public DrawableTypeRequest<String>
load(String string) { return
(DrawableTypeRequest<String>) fromString().load(string); }
```

但是最后都会返回一个DrawableTypeRequest (继承了
DrawableRequestBuilder)

DrawableRequestBuilder就是支持链式调用的一个类,我们平时有类似的需求的时候也可以模仿这样的处理方式,把一些非必须参数用链式调用的方式来设置

四.into方法

```
//GenericRequestBuilder.java    public  
Target<TranscodeType> into(ImageView view) {  
    Util.assertMainThread();      if (view == null) {  
        throw new IllegalArgumentException("You must pass in  
a non null view");      }      if  
(!isTransformationSet && view.getScaleType() != null) {  
        switch (view.getScaleType()) {  
case CENTER_CROP:                  applyCenterCrop();  
        break;                      case FIT_CENTER:  
case FIT_START:                   case FIT_END:  
        applyFitCenter();  
        break;                      // $CASES-OMITTED$  
default:                          // Do nothing.  
        }  
    }  
    return  
into(glide.buildImageViewTarget(view, transcodeClass));  
}
```

这里有三点需要注意的:

- 1.Util.assertMainThread();这里会检查是否主线程,不是的话会抛出异常,所以into方法必须在主线程中调用.
- 2.当你没有调用transform方法,并且你的ImageView设置了ScaleType,那么他会根据你的设置,对图片做处理(具体处理可以查看DrawableRequestBuilder的applyCenterCrop或者applyFitCenter方法,我们自己自定义BitmapTransformation也可以参考这里的处理).
- 3.view在这里被封装成一个Target.

```
//GenericRequestBuilder.java          public <Y extends Target<TranscodeType>> Y into(Y target) {  
    util.assertMainThread();      if (target == null) {  
        throw new IllegalArgumentException("You must pass in a non null Target"); }      if (!isModelSet) {  
        throw new IllegalArgumentException("You must first set a model (try #load())"); }      Request previous =  
    target.getRequest();      if (previous != null) {  
        previous.clear();  
        requestTracker.removeRequest(previous);  
        previous.recycle(); }      Request request =  
    buildRequest(target);      target.setRequest(request);  
    lifecycle.addListener(target);  
    requestTracker.runRequest(request);      return target; }
```

这里可以看到控件封装成的Target能够获取自身绑定的请求,当发现之前的请求还在的时候,会把旧的请求清除掉,绑定新的请求,这也就是为什么控件复用时不会出现图片错位的问题(这点跟我在Picasso源码中看到的处理方式很相像).

接着在into里面会调用buildRequest方法来创建请求

```
//GenericRequestBuilder.java      private Request  
buildRequest(Target<TranscodeType> target) {      if  
(priority == null) {          priority =  
Priority.NORMAL; }      return  
buildRequestRecursive(target, null); }
```

```
//GenericRequestBuilder.java
private Request
buildRequestRecursive(Target<TranscodeType> target,
ThumbnailRequestCoordinator parentCoordinator) {
if (thumbnailRequestBuilder != null) {
    ...
    Request fullRequest = obtainRequest(target,
sizeMultiplier, priority, coordinator);
    ...
    Request thumbRequest =
thumbnailRequestBuilder.buildRequestRecursive(target,
coordinator);
    ...
    coordinator.setRequests(fullRequest, thumbRequest);
    return coordinator;
} else if
(thumbSizeMultiplier != null) {
ThumbnailRequestCoordinator coordinator = new
ThumbnailRequestCoordinator(parentCoordinator);
Request fullRequest = obtainRequest(target,
sizeMultiplier, priority, coordinator);
Request thumbnailRequest = obtainRequest(target,
thumbSizeMultiplier, getThumbnailPriority(),
coordinator);
coordinator.setRequests(fullRequest, thumbnailRequest);
    return coordinator;
} else {
// Base case: no thumbnail.
    return
obtainRequest(target, sizeMultiplier, priority,
parentCoordinator);
}
}
```

1.这里就是请求的生成,buildRequestRecursive里面if有三个分支,这里是根据你设置thumbnail的情况来判断的,第一个是设置缩略图为新的请求的情况,第二个是设置缩略图为float的情况,第三个就是没有设置缩略图的情况.

前两个设置了缩略图的是有两个请求的,fullRequest和thumbnailRequest,没有设置缩略图则肯定只有一个请求了.

2.请求都是通过obtainRequest方法生成的(这个简单了解一下就行)

```
//GenericRequestBuilder.java
private Request obtainRequest(Target<TranscodeType> target, float sizeMultiplier, Priority priority,
RequestCoordinator requestCoordinator) { return
GenericRequest.obtain(...); }
```

REQUEST_POOL是一个队列,当队列中有,那么就从队列中取,没有的话就新建一个GenericRequest

```
//GenericRequest.java
public static <A, T, Z, R>
GenericRequest<A, T, Z, R> obtain(...) {
GenericRequest<A, T, Z, R> request = (GenericRequest<A,
T, Z, R>) REQUEST_POOL.poll(); if (request ==
null) { request = new GenericRequest<A, T, Z,
R>(); } request.init(...); return
request; }
```

回到into方法:当创建了请求后runRequest会调用Request的begin方法,即调用GenericRequest的begin方法

```
//GenericRequestBuilder.java
public <Y extends Target<TranscodeType>> Y into(Y target) { Request
request = buildRequest(target); ... requestTracker.runRequest(request); ... }
```

```
//GenericRequest.java
public void begin() { ...
if (util.isValidDimensions(overrideWidth,
overrideHeight)) { onSizeReady(overrideWidth,
overrideHeight); } else {
target.getSize(this); } ... }
```

最终会调用Engine的load方法

```
//GenericRequest.java
public void onSizeReady(int width,
int height) { ... loadStatus =
engine.load(signature, width, height, dataFetcher,
loadProvider, transformation, transcoder,
priority, isMemoryCacheable, diskCacheStrategy, this);
... }
```

我们先看load方法的前面一段:

- 1.首先会尝试从cache里面取,这里cache就是Glide的构造函数里面的MemoryCache(是一个LruResourceCache),如果取到了,就从cache里面删掉,然后加入activeResources中
- 2.如果cache里面没取到,就会从activeResources中取,activeResources是一个以弱引用为值的map,他是用于存储使用中的资源.之所以在内存缓存的基础上又多了这层缓存,是为了当内存不足而清除cache中的资源中,不会影响使用中的资源.

```
//Engine.java
public <T, Z, R> LoadStatus load(...) {
    ...
    EngineResource<?> cached =
loadFromCache(key, isMemoryCacheable); if (cached
!= null) { cb.onResourceReady(cached);
if (Log.isLoggable(TAG, Log.VERBOSE)) {
logWithTimeAndKey("Loaded resource from cache",
startTime, key); } return null;
}
    EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
if (active != null) {
cb.onResourceReady(active); if
(Log.isLoggable(TAG, Log.VERBOSE)) {
logWithTimeAndKey("Loaded resource from active
resources", startTime, key); }
return null; } ... }
```

load方法接着会通过EngineJobFactory创建一个EngineJob,里面主要管理里两个线程池,diskCacheService和sourceService,他们就是Glide构造函数中Engine里面创建的那两个线程池.

```
//Engine.java public <T, Z, R> LoadStatus load(...) {  
    ...  
    EngineJob engineJob =  
    engineJobFactory.build(key, isMemoryCacheable);  
    ...  
}
```

```
//Engine.java static class EngineJobFactory {  
    private final ExecutorService diskCacheService;  
    private final ExecutorService sourceService;  
    private final EngineJobListener listener;  
    public EngineJobFactory(ExecutorService diskCacheService,  
                           ExecutorService sourceService,  
                           EngineJobListener listener) {  
        this.diskCacheService = diskCacheService;  
        this.sourceService = sourceService;  
        this.listener = listener;  
    }  
    public EngineJob build(Key key,  
                          boolean isMemoryCacheable) {  
        return new EngineJob(key, diskCacheService, sourceService,  
                            isMemoryCacheable, listener);  
    }  
}
```

接着说load方法,前面创建了EngineJob,接着调用EngineJob的start方法,并将EngineRunnable放到diskCacheService(处理磁盘缓存的线程池里面运行),接着线程池就会调用EngineRunnable的run方法.

```
//Engine.java public <T, Z, R> LoadStatus load(...) {  
    ...  
    EngineRunnable runnable = new  
    EngineRunnable(engineJob, decodeJob, priority);  
    jobs.put(key, engineJob);  
    engineJob.addCallback(cb);  
    engineJob.start(runnable);  
    ...  
}
```

```
//EngineJob.java public void start(EngineRunnable  
engineRunnable) {  
    this.engineRunnable =  
    engineRunnable;  
    future =  
    diskCacheService.submit(engineRunnable);  
}
```

```
//EngineRunnable.java
public void run() {
    ...
    try {
        resource = decode();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Exception decoding", e);
        }
        ...
        if (resource == null) {
            onLoadFailed(exception);
        } else {
            onLoadComplete(resource);
        }
    }
}
```

run里面调用的是decode()方法,里面会尝试先从磁盘缓存中读取,如果不从源资源中读取

```
//EngineRunnable.java
private Resource<?> decode()
throws Exception {
    if (isDecodingFromCache()) {
        //第一次会走这
        return decodeFromCache(); //从磁盘缓存中
        //读取
    } else {
        return decodeFromSource(); //从源资
        //源中读取
    }
}
```

我们先来看从磁盘中读取的策略

```
//EngineRunnable.java
private Resource<?>
decodeFromCache() throws Exception {
    Resource<?>
    result = null;
    try {
        result =
        decodeJob.decodeResultFromCache();
    } catch (Exception e) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Exception decoding result from cache: " +
e);
        }
        if (result == null) {
            result =
            decodeJob.decodeSourceFromCache();
        }
    }
    return result;
}
```

我们可以看到这里先尝试读取处理后的图片(Result),然后再尝试读取原图,但是这里面具体逻辑会根据你设置的磁盘缓存策略来决定是否真的会读取处理图和原图

那么我们再回到EngineRunnable的run()方法中

```
public void run() { ... try {
    resource = decode(); } catch
(Exception e) { if (Log.isLoggable(TAG,
Log.VERBOSE)) {
    Log.v(TAG, "Exception
decoding", e); } ...
} else { ...
} }
```

第一次走decode的时候会先尝试从磁盘中获取,如果获取的为null,那么在onLoadFailed方法里面又会把这个run再次放入线程池中,但是这次是放入sourceService(处理源资源的线程池)

```
//EngineRunnable.java private void onLoadFailed(Exception
e) { if (isDecodingFromCache()) { stage =
Stage.SOURCE; manager.submitForSource(this); }
else { manager.onException(e); }}
```

```
//EngineJob.java @Override public void
submitForSource(EngineRunnable runnable) { future =
sourceService.submit(runnable); }
```

接着sourceService里面又会调用调用EngineRunnable的run方法,这次decode里面会走从源资源读取的那条分支

```
//EngineRunnable.java private Resource<?> decode()
throws Exception { if (isDecodingFromCache()) {
//第一次会走这 return decodeFromCache(); //从磁盘缓存中
读取 } else { //第二次会走这 return
decodeFromSource(); //从源资源读取
}} //DecodeJob.java public Resource<z> decodeFromSource()
throws Exception { Resource<t> decoded =
decodeSource(); //获取数据,并解码 return
transformEncodeAndTranscode(decoded); //处理图片
}}
```

里面主要做了两件事,一个是获取图片,一个是处理图片

1. 我们先来看获取图片的decodeSource方法

```
//DecodeJob.java
private Resource<T> decodeSource() throws Exception {
    ...
    //拉取数据
    final A data = fetcher.loadData(priority);
    ...
    //解码，并保存源资源到磁盘
    decoded = decodeFromSourceData(data);
    ...
    return decoded;
}
```

```
//DecodeJob.java
private Resource<T> decodeFromSourceData(A data) throws IOException {
    final Resource<T> decoded;
    if (diskCacheStrategy.cacheSource()) { //解码并保存源资源(图片)到磁盘缓存中
        decoded = cacheAndDecodeSourceData(data);
    } else {
        long startTime = LogTime.getLogTime();
        decoded = loadProvider.getSourceDecoder().decode(data, width, height);
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            logWithTimeAndKey("Decoded from source", startTime);
        }
    }
    return decoded;
}
```

这里调用了DataFetcher的loadData方法来获取数据, DataFetcher有很多实现类, 一般来说我们都是从网络中读取数据, 我们这边就以HttpUrlFetcher为例

想
識
思
學

```
//HttpUrlFetcher.java@Override public InputStream  
loadData(Priority priority) throws Exception { return  
loadDataWithRedirects(glideUrl.toURL(), 0 /*redirects*/,  
null /*lasturl*/, glideUrl.getHeaders()); } private  
InputStream loadDataWithRedirects(URL url, int redirects,  
URL lastUrl, Map<String, String> headers) throws  
IOException { if (redirects >= MAXIMUM_REDIRECTS) {  
    throw new IOException("Too many (> " +  
    MAXIMUM_REDIRECTS + ") redirects!"); } else {  
    // Comparing the URLs using .equals performs additional  
    // network I/O and is generally broken. // See  
    // http://michaelscharf.blogspot.com/2006/11/javaneturlequal  
    // s-and-hashcode-make.html. try { if  
    (lastUrl != null && url.toURI().equals(lastUrl.toURI()))  
    { throw new IOException("In re-direct  
    loop"); } } catch (URISyntaxException  
e) { // Do nothing, this is best effort.  
} urlConnection = connectionFactory.build(url);  
for (Map.Entry<String, String> headerEntry :  
headers.entrySet()) {  
    urlConnection.addRequestProperty(headerEntry.getKey(),  
    headerEntry.getValue()); }  
urlConnection.setConnectTimeout(2500);  
urlConnection.setReadTimeout(2500);  
urlConnection.setUseCaches(false);  
urlConnection.setDoInput(true); // Connect explicitly  
// to avoid errors in decoders if connection fails.  
urlConnection.connect(); if (isCancelled) {  
    return null; } final int statusCode =  
urlConnection.getResponseCode(); if (statusCode / 100  
== 2) { //请求成功 return  
getStreamForSuccessfulRequest(urlConnection); } else  
if (statusCode / 100 == 3) { String  
redirect urlString =  
urlConnection.getHeaderField("Location"); if  
(TextUtils.isEmpty(redirect urlString)) { throw  
new IOException("Received empty or null redirect url");  
} URL redirectUrl = new URL(url,
```

```
redirectUrlString); return
loadDataWithRedirects(redirectUrl, redirects + 1, url,
headers); } else { if (statusCode == -1) {
    throw new IOException("Unable to retrieve response
code from HttpURLConnection."); } throw new
IOException("Request failed " + statusCode + ": " +
urlConnection.getResponseMessage()); }}
```

2. 看完了获取图片的方法, 我们再来看看处理图片的方法

transformEncodeAndTranscode

```
//DecodeJob.java private Resource<Z>
transformEncodeAndTranscode(Resource<T> decoded) { ...
//对图片做剪裁等处理 Resource<T> transformed =
transform(decoded); ... //将处理后的图片写入磁盘缓存(会根据
配置来决定是否写入) writeTransformedToCache(transformed);
... //转码, 转为需要的类型 Resource<Z> result =
transcode(transformed); ... return result; }
```

Glide的整个加载流程就基本上走完了, 整个篇幅还是比较长的, 第一次看得时候可能会有点懵逼, 需要结合着源码多走几遍才能够更加熟悉.

阅读源码并不是我们最终的目的, 我们阅读源码主要有两个目的

一个是能够更加熟悉这个框架, 那么使用的时候就更加得心应手了, 比如我从源码中发现了很多文章都说错了默认的缓存策略

一个是学习里面的设计模式和思想, 应用到我们自己的项目中, 比如说面对接口编程, 对于不同的数据, 用DataFetcher的不同实现类来拉取数据

3.26 Glide ‘优’ 与 ‘愁’

随着业务的增长, 加载图片是不可避免的需求。从一开始的自己写一个 ImageLoader 到井喷似的第三方图片加载库, 当然中间还时不时穿插着 asynctask, 三级缓存, LRU Cache 等。那个时候想必大家都用过 nostra13/Android-Universal-Image-Loader, 大家都纷纷拥有了它, 确实时势造英雄吧, 所以我很钦佩作者。但三年前作者大概可能

觉得实在是太累了的，宣布不再维护了的。在此期间也有一些优秀的开源库比如 square 出来的 Picasso ,优雅的链式调用想必很多人选择了拥抱他。后来Google在2014年的google I/O大会上发布的官方app中使用的 bumptech/glide 闯入大家的视野， Google 推荐大家图片加载使用 Glide. 当然 Glide 的使用方式也是仿照 Picasso 。所以几乎没有任何迁移成本，很多人也开始拥抱了 Glide. 当然在此期间 Facebook 也不甘寂寞横空出来开源了 fresco 。

为什么会选择 Glide

为什么选择 Glide ，前言中也提到了毕竟是 Google 推荐的最佳选择。除此之外也可以做一下简单的对比

Glide VS Picasso

双胞胎兄弟之间的对比，使用方式相同，但 Glide 之所以胜出，不仅仅是 Google 的推荐，更多应该归功于 GIF 的支持。在没有 Glide 之前，常用的做法就是写了个自定义 view 然后用一个 media 去播放。有了 Glide 之后几乎对于 GIF 无感知了的，内部已经支持了的。可以像普通图片那样去加载并且显示出来动图。

Glide VS Android-Universal-Image-Loader

虽然有再多的不舍，一个已经不再维护的开源库，Android碎片化那么严重，我们自己维护起来还是要考虑成本的。所以 Glide 胜出。

Glide VS fresco

两个都支持 GIF。所以 GIF 这一关pass掉。说到这里不得不提到一个头疼的OOM问题，fresco 之所以很快闯入大家的视线，大概就是因为 Facebook 说他们使用了 native 内存规避掉了 OutOfMemoryError 问题。而且官方还专门写了个demo，把几大流行的开源库都集成进去，为了说明自己的图片加载库加载同样的图片速度更快，内存占用更低。所以 fresco 相比较于 Glide 的（官方）优势就是这两点：内存以及加载速度。但是我为什么依旧坚持抛弃了 fresco ？

1. “In Android 4.x and lower, Fresco puts images in a special region of Android memory. This lets your application run faster - and suffer the dreaded OutOfMemoryError much less often.” 官方的原话是这么说的，所以在高版本上面依旧使用的Java 内存，所以不可避免依旧会占用内存。
2. 提到内存，不得不说到另外一个笑话，fresco 最大只支持图片文件大小为 2M 。记得有一次帮其他团队跟踪问题，看到了 fresco 源码中有一个最大 size 2M 常量。于是当场找了一个10M的图片作为测试。 Glide 正常显示， fresco显示黑屏。。。
3. 使用方式上，fresco 推荐的是用他提供的 SimpleDraweeView . 这个方式意味着我们的迁移成本会非常的高，要改布局文件，其次还必须给定大小（或者比例）。当然他也支持代码来加载图片，比如 DraweeHierarchy，但是写起来还是真心很费劲的，很不友好，改动成本居高。
4. fresco 更多是native实现。所以需要对NDK有所了解，但个人对NDK不太了解，相比较于 Glide， 同样遇到问题之后，修改源码的成本， Glide 成本更可控。前者可能就不太好下手了的。
5. Glide 各种 BitmapTransformation，比如圆形，圆角等，更让人喜欢。
6. 这一点就当随意吐槽一下，当然也可以说心疼一下 Facebook。因为在没有 Android studio (gradle构建) 的情况下，想必大家都用的是 eclipse 吧。那么就意味着 fresco 得提供 Jar 包. 这一点当时也是把很多人拒之门外了的，可笑的是当 Facebook 费了老大劲的搞出来 jar 包之后，大家早就纷纷转战 gradle 构建工程, 直接 maven 依赖啦。大写的尴尬。

综上所述，Glide 依旧胜出。

Glide 是如何解决图片加载生命周期的？（精髓之一）（也是bug高发地带）

当一个界面离开之后，我们更希望当前的图片取消加载，那么 Glide 是怎么做到的呢？

Glide 的使用方式上，一定需要传入一个 context 给它。它为什么需要拿上下文呢？原因就是可以根据不同的上下文进行处理，拿到 context（除了 application context）之后，Glide 做了一件很巧妙的事情，就是在这个界面上追加一个 fragment，由于 fragment 添加到了 activity 上，是可以捕获到生命周期的，因此可以在 destroy 的时候取消掉当前 context 下的 glide 对象中的加载任务。

为什么标题后面说是‘也是 bug 高发地带’呢？因为从实现方式上，它是巧妙的利用了 fragment 的生命周期来实现的‘销毁’动作，那么就类似于另外一个高发 bug，延时的匿名内部类（网络请求 callback 回来），界面已经销毁，所以当前 activity 依附的 glide 也就销毁了的，此时再尝试加载图片的话，就会 crash。具体源码中可以看到这里：

<https://github.com/bumptech/glide/blob/master/library/src/main/java/com/bumptech/glide/Glide.java>

```
@NotNull private static RequestManagerRetriever  
getRetriever(@Nullable Context context) { // Context  
could be null for other reasons (ie the user passes in  
null), but in practice it will // only occur due to  
errors with the Fragment lifecycle.  
Preconditions.checkNotNull(context, "You  
cannot start a load on a not yet attached view or a  
Fragment where getActivity() " + "returns null  
(which usually occurs when getActivity() is called before  
the Fragment " + "is attached or after the  
Fragment is destroyed."); return  
Glide.get(context).getRequestManagerRetriever(); }
```

Glide 是如何做到链式的优雅调用？（精髓之二）

链式调用其实只不过是一种语法招数。它能让你通过重用一个初始操作来达到用少量代码表达复杂操作的目的。其实就是类似 builder 一样，把 this return 出去，一直可以调用自己的方法。所以也称之为方法链。

```
// Some time in the future: Glide.with(fragment)  
.load(newUrl) .into(target);
```

链式调用的好与不好：

1. 编程性强
2. 可读性强
3. 代码简洁
4. 对程序员的业务能力要求高
5. 不太利于代码调试

官方说了的，不支持并且不建议imageview设置wrap_content。因为这样 glide 不知道要加载多大的图片给我们才好，在他的接口（Sizes and dimensions）中也有体现。普通的imageview其实也还好，如果放在列表（RecyclerView）中，由于我们并不知道目标图片大小是多大的，所以我们选择了wrap_content，那么在上下来回滚动过程中，就会导致图片一会大一会小的bug.

官方 issue 作者回答如下：

Don't use wrap_content. Even if you don't use Glide, wrap_content necessarily means that the size of your views in RecyclerView are going to change from item to item. That's going to cause all sorts of UI weirdness. One option is to try to obtain the image dimensions in whatever metadata you're using to populate the RecyclerView. Then you can set a fixed view size in onBindViewHolder so the view size at least doesn't change when the image is loaded. You're still likely to see weird scroll bar behavior though. If nothing else, you can always pick a uniform size that's large enough for all items and use the same consistent size for every item. For the image file size, you can downscale or upscale by setting the ImageView size manually to 150dp x 150dp. Ultimately either you need uniform view sizes or predetermined view sizes. There's nothing else that will prevent content from expanding or shrinking in your RecyclerView. For the placeholder bit, I think that will be fixed by 648c58e, you can check by trying the 4.2.0-SNAPSHOT version:

<http://bumptech.github.io/glide/dev/snapshots.html>.

so...还是建议我们指定图片的大小。

Glide 坑爹的 support包 版本问题

为什么会有这个问题呢？其实刚才已经提到了的，由于它用到了 fragment，那么自然就有版本冲突问题。support 包大家都懂的，不同的版本，差异可能巨大，有个段子就是说 Google 的 support 包大概是招了个实习生写的。不同的版本冲突可能会编译不过，可能会有 ‘nosuchmethod’ 等等问题。

比如我们产线现在的用的是 Glide 版本是 4.3.1，之所以迟迟没有升级到最新版本，就是因为后面的版本 Glide 采用了 27 编译。。。而我们项目才 25 ... 中间这个编译升级的风险。有点不可控。所以一直没有升级上去。所以建议，在升级 Glide 版本的时候看一下对应版本源码中依赖的 support 版本是多少。

写在最后

之所以今天简单的跟大家聊一聊 Glide。其实也只是找了一个项目中用到的开源库作为例子，想跟大家聊聊，当项目中需要技术选型的时候，不能给的答案是：因为大家都在用啊？

而我更想知道的是，大家为什么会选择它，不仅仅是人群中多看了你一眼，而是从外表 API 的“美”，再到内在框架设计的“美”。只有知其所以然，那么当遇到坑的时候，才知道如何去解决它。而不是简单的“跟风似的”“一见钟情”。。。

3.27 关于图片加载神器--Glide与Picasso的使用与比较

什么是Glide?

Glide 是一个加载图片的库，作者是bumptech，它是在泰国举行的 google 开发者论坛上 google 为我们介绍的，这个库被广泛的运用在 google 的开源项目中。

Glide解决什么问题？

Glide是一个非常成熟的图片加载库，他可以从多个源加载图片，如：网络，本地，Uri等，更重要的是他内部封装了非常好的缓存机制并且在处理图片的时候能保持一个低的内存消耗。

Glide怎么使用？

在Glide的使用方面，它和Picasso的使用方法是比较相似的，并且他们的运行机制也有很多相似的地方，很多博文会把两者进行比较，此文也采用同样的方式，通过比较两者来学习他们之间的优点和不足。

首先，当我们使用这两个库的时候第一步要做的就是导入库，Picasso好说，直接依赖就行，但是Glide要注意，这个库是要依赖于support library v4的，所以用这个库的时候，不要忘了依赖v4包。

基本使用

在基本使用方面这两个库非常的相似，如下代码所示：

Picasso：

```
Picasso.with(context)
    .load("http://inthecheesefactory.com/uploads/source/glide
picasso/cover.jpg")    .into(ivImg);
```

Glide：

```
Glide.with(context)
    .load("http://inthecheesefactory.com/uploads/source/glide
picasso/cover.jpg")    .into(ivImg);
```

看到没有，表面上看是不是非常相似，其实他们有一个不一样的地方，就是Picasso的with只能传入context，而Glide的with可以传入context，还可以是Activity或者是Fragment，你可能会问，这有什么用呢？用处就是图片的加载可以和Activity或者Fragment保持一致，不至于出现，Activity已经暂停了，但是图片却还在加载的情况。

默认的Bitmap格式是RGB_565

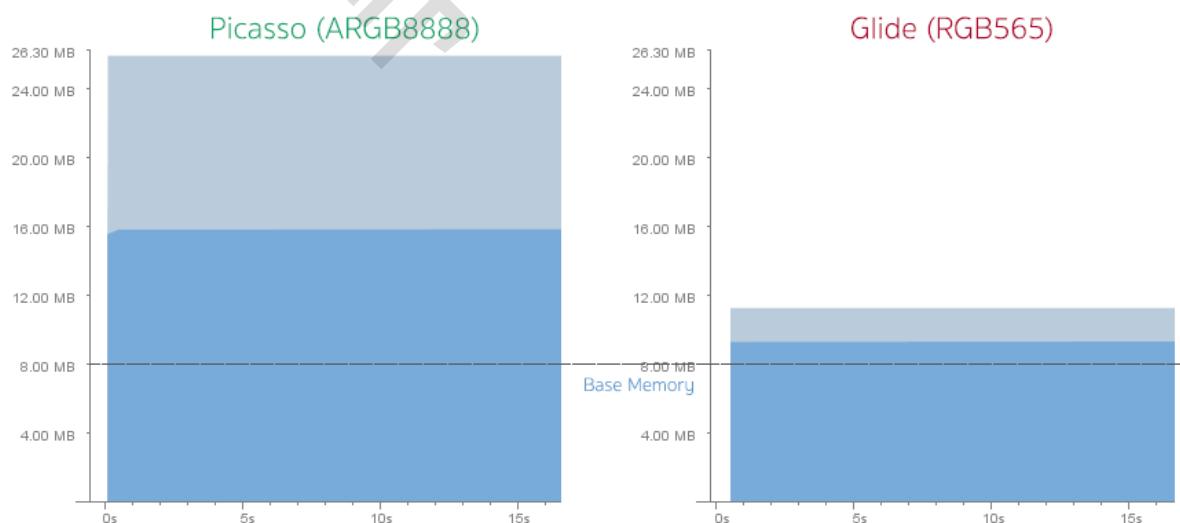
一下是Picasso和Glide加载后的结果(1920x1080 像素的图片被加载到768x432像素的imageview中)：



你可以看到，被Glide加载的图片在质量上不如Picasso加载的图片，这是为什么？其实是因为Glide的Bitmap默认的格式是RGB_565,而Picasso用的是ARGB_8888,所以虽然质量上不如Picasso(其实在手机上也不明显),但是RGB_565格式的图片仅仅消耗ARGB_8888格式图片一半的内存。

Here is the memory consumption graphs between Picasso at ARGB8888 and Glide at RGB565. (Base application consumes around 8MB)

下图是Picass的ARGB8888格式图片和Glide的RGB565格式图片的内存消耗比较（应用本身大约占8M）：



如果你对图片没有过高的要求，那么用默认的格式就可以，但是如果你对图片质量要求较高，那么可以把图片的格式转换为ARGB8888，方法就是通过继承GlideModule,如下所示：

```
public class GlideConfiguration implements GlideModule {  
    @Override public void applyOptions(Context context,  
        GlideBuilder builder) { // Apply options to the  
        // builder here.  
        builder.setDecodeFormat(DecodeFormat.PREFER_ARGB_8888);  
    } @Override public void registerComponents(Context  
        context, Glide glide) { // register ModelLoaders  
        // here. }}
```

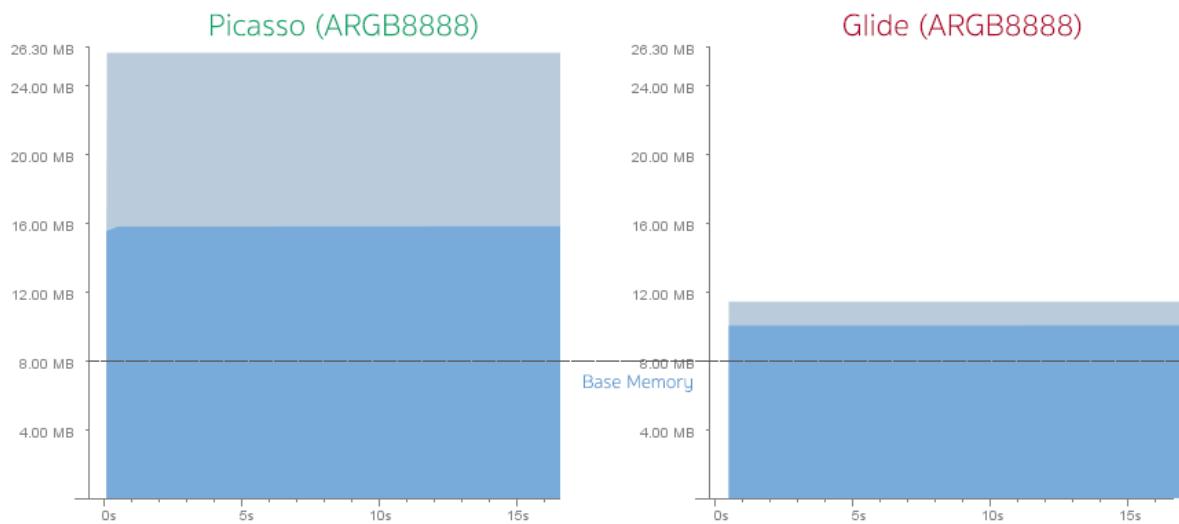
然后在AndroidManifest.xml中进行定义：

```
<meta-data  
    android:name="com.inthecheesefactory.lab.glidepicasso.Gli  
deConfiguration" android:value="GlideModule"/>
```

在次看下图，是不是完全一样了呢？



那让我们再看一看两者之间的内存消耗：



我们发现，虽然用的图片格式是一样的，并且Glide加载的几乎是先前的两倍内存，但是Picasso消耗的内存仍然远大于Glide.

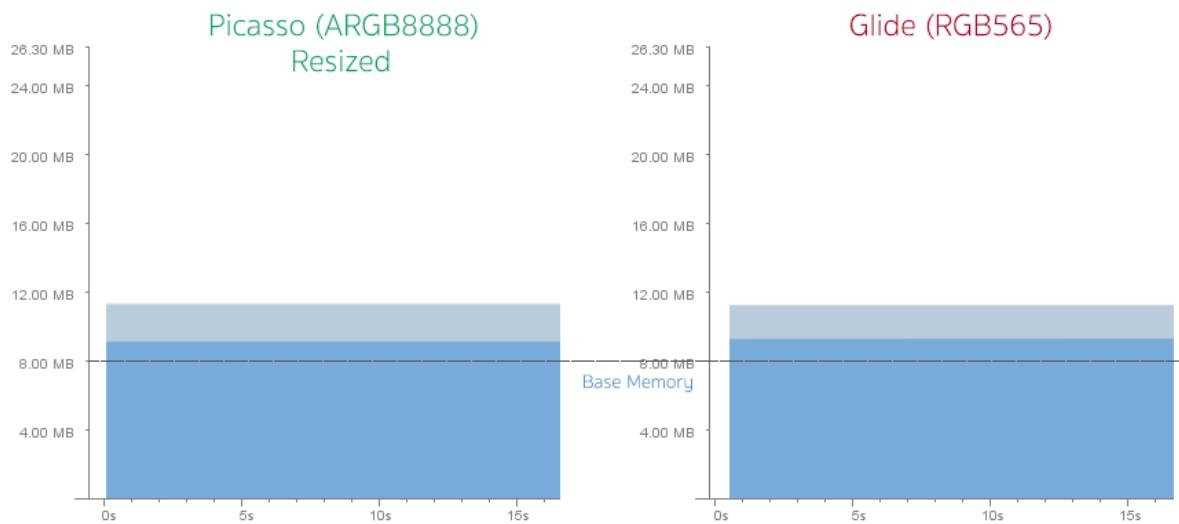
这是因为，Picasso加载了完整尺寸的图片（1920x1080像素）进入内存，当绘图的时候，让GPU即时的恢复到所需要的尺寸（768x432像素），然而Glide则加载精确的imageview尺寸进入内存，当然，我们可以手动使Picasso也使用这种方式加载图片：

```
Picasso.with(this)
    .load("http://nuuneoi.com/uploads/source/playstore/cover.
jpg")      .resize(768, 432)      .into(ivImgPicasso);
```

上面的方式有局限性，就是我们必须知道imageview的精确尺寸，而如果我们的imageview设置了wrap，我们就不能用上面的那种方式了，而需要改为下面的方式：

```
Picasso.with(this)
    .load("http://nuuneoi.com/uploads/source/playstore/cover.
jpg")      .fit()      .centerCrop()      .into(ivImgPicasso);
```

现在，我们再来看内存消耗图：



哈哈，两者的内存消耗现在差不多一样了，但是不得不说的是在这一点上 Glide确实比Picasso做的要好，因为Glide可以在每种情况下自动的计算 ImageView的尺寸。

图片的质量细节

当我把imageview的尺寸调整到和图片一样大的时候（1920x1080像素），我们来观察一下下面的图片：



这次两张图片的对比就比较明显了， Glide加载的图片可以明显的看到锯齿像素点，但是当用户使用应用的时候，这并不是那么容易察觉，并且，如果真的忍受不了这种小瑕疵，可以把图片格式调整到ARGB_8888。

外部缓存

在默认情况下Picasso和Glide的外部缓存机制是非常不一样的，通过实验可以发现（1920x1080像素的图片被加载到768x432像素的imageview中），Glide缓存的是768x432像素的图片，而Picasso缓存的是整张图片（1920x1080像素）。



如果加载的图片是RGB565模式，缓存图片也是RGB565模式。

当我们调整imageview的大小时，Picasso会不管imageview大小是什么，总是直接缓存整张图片，而Glide就不一样了，它会为每个不同尺寸的Imageview缓存一张图片，也就是说不管你的这张图片有没有加载过，只要imageview的尺寸不一样，那么Glide就会重新加载一次，这时候，它会在加载的imageview之前从网络上重新下载，然后再缓存。

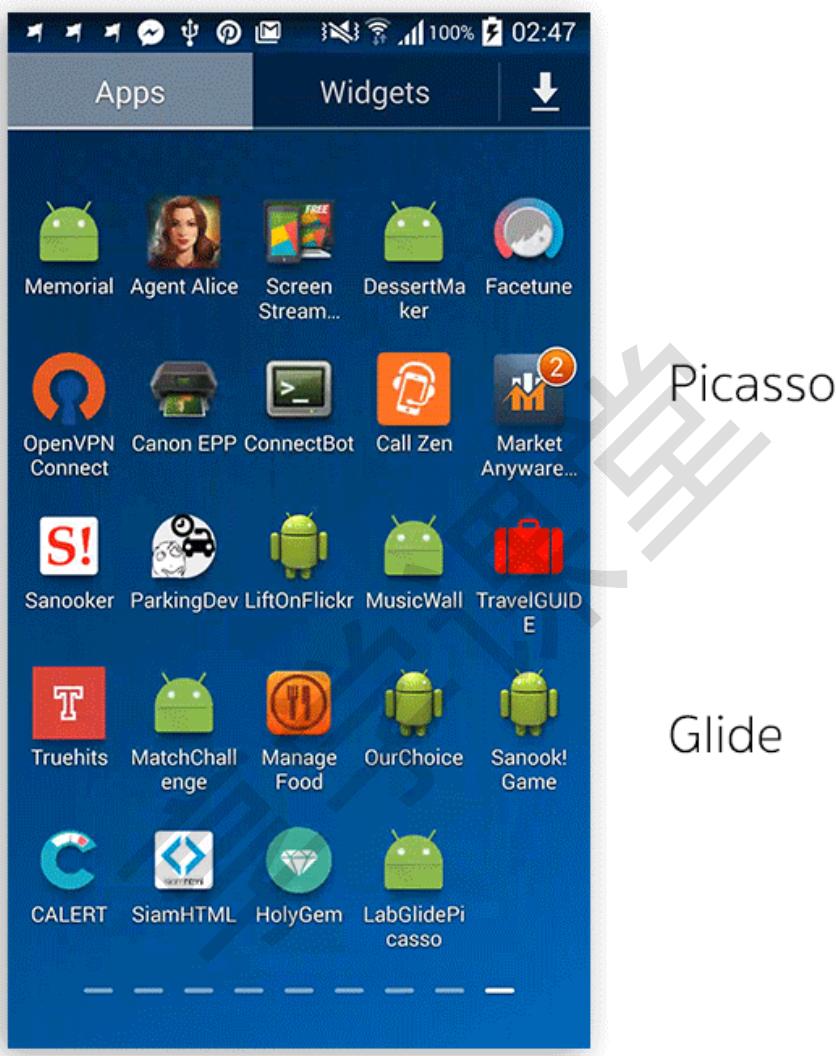
防止各位不明白，再来举个例子，如果一个页面的imageview是200*200像素，而另一个页面中的imageview是100*100像素，这时候想要让两个imageview像是同一张图片，那么Glide需要下载两次图片，并且缓存两张图片。

但是我们可以通过如下的方法来让Glide即缓存全尺寸的图片，有缓存不同尺寸的图片：

```
Glide.with(this)
.load("http://nuuneoi.com/uploads/source/playstore/cover.jpg")
.diskCacheStrategy(DiskCacheStrategy.ALL)
.into(ivImgGlide);
```

Glide的这种默认的缓存机制有一个优点，就是它可以加快图片加载的速度（可以理解为以空间换时间），而Picasso会造成一定的延迟，因为它在加载到imageview的时候，总是需要调整大小，确实Picasso有一个立即显示图片的方法（如下所示），但是这还是消除不了延时。

```
//Picasso.noFade();
```



在外部缓存方面Glide和Picasso各有所长，你可以选择合适自己的来用（也就是对于你的app来说是控件重要还是时间重要）。

特性

你几乎可以用Glide来做Picasso可以做的所有事情，并且他们的代码风格也非常类似：

调整图片尺寸：

```
// Picasso.resize(300, 200); // Glide.override(300, 200);
```

Center Cropping: 中心裁剪

```
// Picasso.centerCrop(); // Glide.centerCrop();
```

图形变换 (Transforming) :

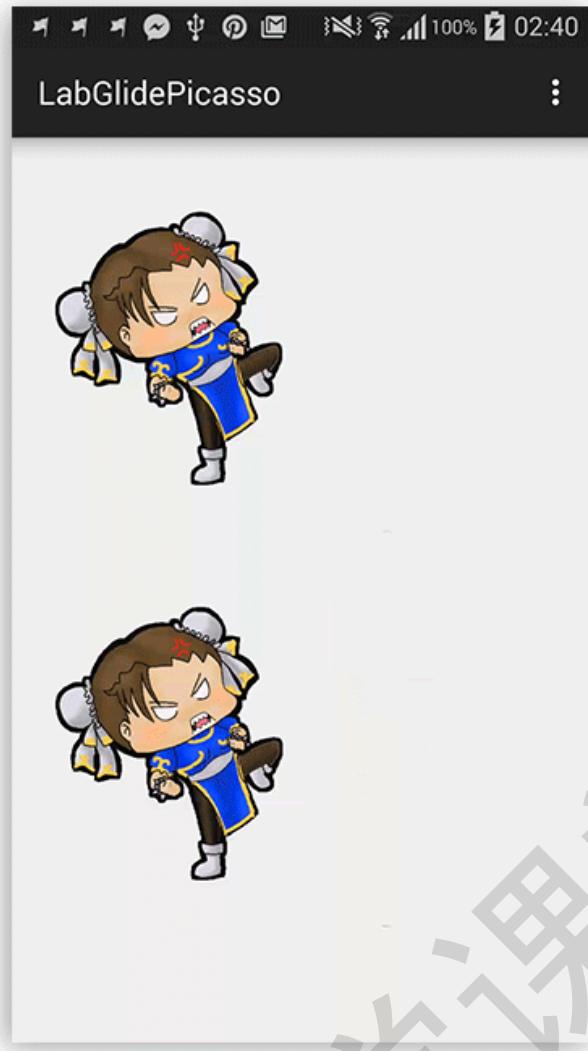
```
// Picasso.transform(new CircleTransform()) //  
Glide.transform(new CircleTransform(context))
```

设置占位图片和错误图片:

```
//  
Picasso.placeholder(R.drawable.placeholder).error(R.drawable.  
imagenotfound) //  
Glide.placeholder(R.drawable.placeholder).error(R.drawable.  
imagenotfound)
```

有什么Glide可以做Picasso却做不了

Glide的一个明显的优点就是它可以加载gif图片，你可能说我用Picasso加载也不报错啊？你要注意，用Picasso加载的gif图片是不会动的，如下所示：



因为Glide被设计成能和Activity/Fragment的生命周期完美的相结合，因此gif动画将随着Activity/Fragment的生命周期自动的开始和停止。

gif的缓存和一般的图片也是一样的，也是第一次加载的时候调整大小，然后缓存。

但是，要注意的是，通过测量，我们可以发现gif图片将消耗非常多的内存，因此使用它的时候要慎重。

除了加载gif图片外，Glide还可以解析任何的video文件成为一个静态图片。

另一个比较有用的特性是，你可以配置显示图片的动画，而Picasso只支持一个淡入(fading in)动画效果。

你也可以用`thumbnail()`来创造一个image的thumbnail(极小)的图片。

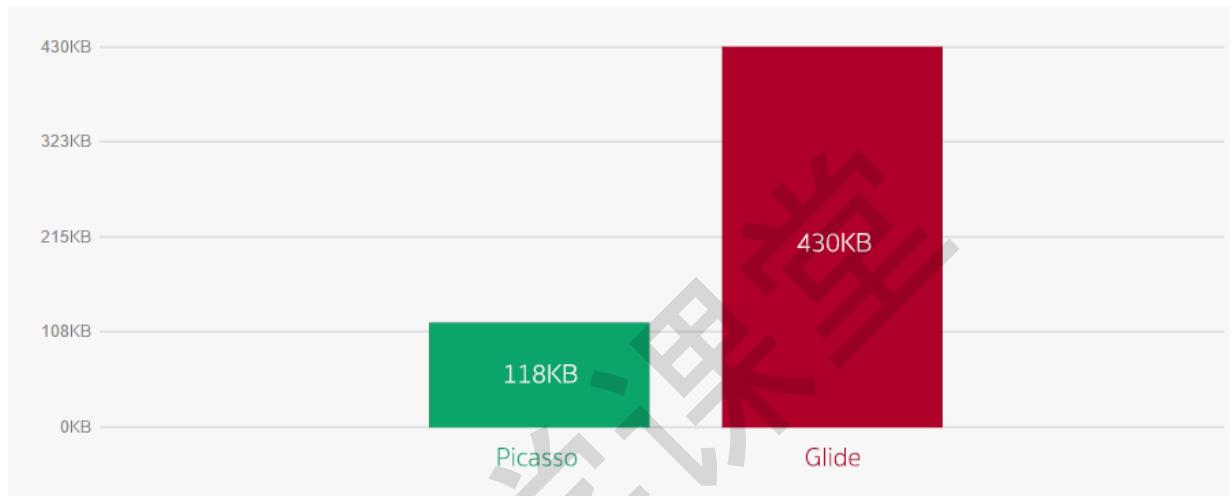
还有很多的特性，但是一般都不太常用，如，把一个图片的编码转换为字节数组，等。

配置

我们可以对很多的配置做出调整，如，外部缓存的大小和位置，内部缓存的最大限制，Bitmap的格式等等，至于更多的配置，可以参考[配置页面](#)。

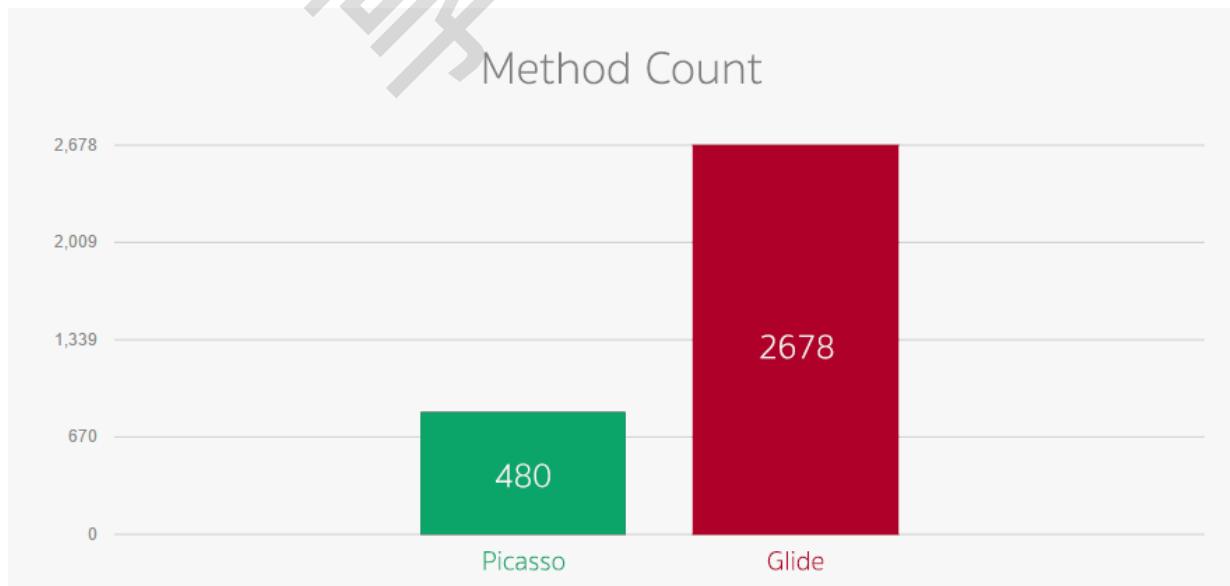
库的大小

Picasso的大小大约是118KB，而Glide大约有430KB。



一个库的大小有什么意义吗？确实，我也认为意义不大！

我们再来看一下两者之间的方法数量的比较：



值得注意的是，在[Android](#) DEX file中的方法是有限制的，最大方法数为65535个，从这一点来说，Glide的方法确实不少，并且，混淆器也建议对我们的项目进行混淆。

总结

Glide和Picasso都不是完美的，从某些方面来说，Glide在图片的缓存上来说是比较不错的，因为它的速度比较快，另外，它也可以有效的防止OOM错误，而加载gif图片也是Glide的一大优势，但是默认情况下picasso的图片质量是很高的。

另外的一点小建议是，使用Glide的时候把图片的格式改为ARGB8888并且缓存全尺寸和其他尺寸的图片，这样使用可以让加载图片更好。

<https://inthecheesefactory.com/blog/get-to-know-glide-recommended-by-google/en>

3.28 Q1：看过Glide源码吗，你印象最深的是什么？

Glide缓存简介

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就分别来分析一下这两种缓存的使用方法以及它们的实现原理。

3.29 Q2：简单说一下Glide的三级缓存？

Glide缓存简介

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了Glide极佳的图片缓存效果，那么接下来我们就分别来分析一下这两种缓存的使用方法以及它们的实现原理。

缓存Key

既然是缓存功能，就必然会有用于进行缓存的Key。那么Glide的缓存Key是怎么生成的呢？我不得不说，Glide的缓存Key生成规则非常繁琐，决定缓存Key的参数竟然有10个之多。不过繁琐归繁琐，至少逻辑还是比较简单的，我们先来看一下Glide缓存Key的生成逻辑。

生成缓存Key的代码在Engine类的load()方法当中，这部分代码我们在上一篇文章当中已经分析过了，只不过当时忽略了缓存相关的内容，那么我们现在重新来看一下：

```
public class Engine implements EngineJobListener,
MemoryCache.ResourceRemovedListener,
EngineResource.ResourceListener {    public <T, Z, R>
LoadStatus load(Key signature, int width, int height,
DataFetcher<T> fetcher,                      DataLoadProvider<T, Z>
LoadProvider, Transformation<Z> transformation,
ResourceTranscoder<Z, R> transcoder,          Priority
Priority priority, boolean isMemoryCacheable, DiskCacheStrategy
diskCacheStrategy, ResourceCallback cb) {
util.assertMainThread();      long startTime =
LogTime.getLogTime();      final String id =
fetcher.getId();            EngineKey key =
keyFactory.buildKey(id, signature, width, height,
LoadProvider.getCacheDecoder(),
LoadProvider.getSourceDecoder(), transformation,
LoadProvider.getEncoder(),                  transcoder,
LoadProvider.getSourceEncoder());           ...     ...}
```

可以看到，这里在第11行调用了fetcher.getId()方法获得了一个id字符串，这个字符串也就是我们要加载的图片的唯一标识，比如说如果是一张网络上的图片的话，那么这个id就是这张图片的url地址。

接下来在第12行，将这个id连同着signature、width、height等等10个参数一起传入到EngineKeyFactory的buildKey()方法当中，从而构建出了一个EngineKey对象，这个EngineKey也就是Glide中的缓存Key了。

可见，决定缓存Key的条件非常多，即使你用override()方法改变了一下图片的width或者height，也会生成一个完全不同的缓存Key。

EngineKey类的源码大家有兴趣可以自己去看一下，其实主要就是重写了equals()和hashCode()方法，保证只有传入EngineKey的所有参数都相同的情况下才认为是同一个EngineKey对象，我就不在这里将源码贴出来了。

内存缓存：

详细讲解：

1、LruCache算法

LruCache算法，又称为近期最少使用算法。主要算法原理就是把最近所使用的对象的强引用存储在LinkedHashMap上，并且，把最近最少使用的对象在缓存池达到预设值之前从内存中移除。

2、弱引用

有了缓存Key，接下来就可以开始进行缓存了，那么我们先从内存缓存看起。

首先你要知道，默认情况下，Glide自动就是开启内存缓存的。也就是说，当我们使用Glide加载了一张图片之后，这张图片就会被缓存到内存当中，只要在它还没从内存中被清除之前，下次使用Glide再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比方说你在一个RecyclerView当中反复上下滑动，RecyclerView中只要是Glide加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而Glide最为人性化的是，你甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为Glide默认就已经将它开启了。

那么既然已经默认开启了这个功能，还有什么可讲的用法呢？只有一点，如果你有什么特殊的原因需要禁用内存缓存功能，Glide对此提供了接口：

```
Glide.with(this).load(url)
    .skipMemoryCache(true).into(imageview);
```

可以看到，只需要调用skipMemoryCache()方法并传入true，就表示禁用掉Glide的内存缓存功能。

没错，关于Glide内存缓存的用法就只有这么多，可以说是相当简单。但是我们不可能只停留在这么简单的层面上，接下来就让我们就通过阅读源码来分析一下Glide的内存缓存功能是如何实现的。

其实说到内存缓存的实现，非常容易就让人想到LruCache算法（Least Recently Used），也叫近期最少使用算法。它的主要算法原理就是把最近使用的对象用强引用存储在LinkedHashMap中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。

那么不必多说，Glide内存缓存的实现自然也是使用的LruCache算法。不过除了LruCache算法之外，Glide还结合了一种弱引用的机制，共同完成了内存缓存功能，下面就让我们来通过源码分析一下。

首先回忆一下，在上一篇文章的第二步load()方法中，我们当时分析到了在loadGeneric()方法中会调用Glide.buildStreamModelLoader()方法来获取一个ModelLoader对象。当时没有再跟进到这个方法的里面再去分析，那么我们现在来看下它的源码：

```
public class Glide {    public static <T, Y>  
ModelLoader<T, Y> buildModelLoader(Class<T> modelClass,  
Class<Y> resourceClass, Context context) {  
    if (modelClass == null) {                if  
(Log.isLoggable(TAG, Log.DEBUG)) {  
Log.d(TAG, "Unable to load null model, setting  
placeholder only");            }           return null;  
    }           return  
Glide.get(context).getLoaderFactory().buildModelLoader(mo  
delClass, resourceClass);    }    public static Glide  
get(Context context) {        if (glide == null) {  
    synchronized (Glide.class) {                if  
(glide == null) {  
        Context  
applicationContext = context.getApplicationContext();  
        List<GlideModule> modules = new  
ManifestParser(applicationContext).parse();  
        GlideBuilder builder = new  
GlideBuilder(applicationContext);  
        for  
(GlideModule module : modules) {  
    module.applyOptions(applicationContext, builder);  
    }           glide =  
builder.createGlide();  
        for  
(GlideModule module : modules) {  
    module.registerComponents(applicationContext, glide);  
    }           }           }  
    return glide;    }    ...}  
}
```

这里我们还是只看关键，在第11行去构建ModelLoader对象的时候，先调用了一个Glide.get()方法，而这个方法就是关键。我们可以看到，get()方法中实现的是一个单例功能，而创建Glide对象则是在第24行调用GlideBuilder的createGlide()方法来创建的，那么我们跟到这个方法当中：

```
public class GlideBuilder {    ...    Glide createGlide()  
{        if (sourceService == null) {            final  
int cores = Math.max(1,  
Runtime.getRuntime().availableProcessors());  
sourceService = new  
FifoPriorityThreadPoolExecutor(cores);        }        if  
(diskCacheService == null) {            diskCacheService  
= new FifoPriorityThreadPoolExecutor(1);        }  
MemorySizeCalculator calculator = new  
MemorySizeCalculator(context);        if (bitmapPool ==  
null) {            if (Build.VERSION.SDK_INT >=  
Build.VERSION_CODES.HONEYCOMB) {                int size  
= calculator.getBitmapPoolSize();  
bitmapPool = new LruBitmapPool(size);            } else {  
                bitmapPool = new BitmapPoolAdapter();  
            }        }        if (memoryCache == null) {  
            memoryCache = new  
LruResourceCache(calculator.getMemoryCacheSize());  
        }        if (diskCacheFactory == null) {  
            diskCacheFactory = new  
InternalCacheDiskCacheFactory(context);        }  
if (engine == null) {            engine = new  
Engine(memoryCache, diskCacheFactory, diskCacheService,  
sourceService);        }        if (decodeFormat == null)  
{            decodeFormat = DecodeFormat.DEFAULT;  
        }        return new Glide(engine, memoryCache,  
bitmapPool, context, decodeFormat);    }}}
```

这里也就是构建Glide对象的地方了。那么观察第22行，你会发现这里new出了一个LruResourceCache，并把它赋值到了memoryCache这个对象上面。你没有猜错，这个就是Glide实现内存缓存所使用的LruCache对象了。不过我这里并不打算展开来讲LruCache算法的具体实现，如果你感兴趣的话可以自己研究一下它的源码。

现在创建好了LruResourceCache对象只能说是把准备工作做好了，接下来我们就一步步研究Glide中的内存缓存到底是如何实现的。

刚才在Engine的load()方法中我们已经看到了生成缓存Key的代码，而内存缓存的代码其实也是在这里实现的，那么我们重新来看一下Engine类load()方法的完整源码：

阅读理解

想
識
思
學

```
public class Engine implements EngineJobListener,
MemoryCache.ResourceRemovedListener,
EngineResource.ResourceListener { ... public
<T, Z, R> LoadStatus load(Key signature, int width, int
height, DataFetcher<T> fetcher,
DataLoadProvider<T, Z> loadProvider, Transformation<Z>
transformation, ResourceTranscoder<Z, R> transcoder,
Priority priority, boolean isMemoryCacheable,
DiskCacheStrategy diskCacheStrategy, ResourceCallback cb)
{ Util.assertMainThread(); long startTime =
LogTime.getLogTime(); final String id =
fetcher.getId(); EngineKey key =
keyFactory.buildKey(id, signature, width, height,
loadProvider.getCacheDecoder(),
loadProvider.getSourceDecoder(), transformation,
loadProvider.getEncoder(), transcoder,
loadProvider.getSourceEncoder()); EngineResource<?
> cached = loadFromCache(key, isMemoryCacheable);
if (cached != null) {
cb.onResourceReady(cached); if
(Log.isLoggable(TAG, Log.VERBOSE)) {
logWithTimeAndKey("Loaded resource from cache",
startTime, key); } return null;
} EngineResource<?> active =
loadFromActiveResources(key, isMemoryCacheable);
if (active != null) {
cb.onResourceReady(active); if
(Log.isLoggable(TAG, Log.VERBOSE)) {
logWithTimeAndKey("Loaded resource from active
resources", startTime, key); }
return null; } EngineJob current =
jobs.get(key); if (current != null) {
current.addCallback(cb); if
(Log.isLoggable(TAG, Log.VERBOSE)) {
logWithTimeAndKey("Added to existing load", startTime,
key); } return new LoadStatus(cb,
current); } EngineJob engineJob =
engineJobFactory.build(key, isMemoryCacheable);
```

```
DecodeJob<T, Z, R> decodeJob = new DecodeJob<T, Z, R>
(key, width, height, fetcher, loadProvider,
transformation, transformation, transformation,
diskCacheProvider, diskCacheStrategy, priority);
EngineRunnable runnable = new EngineRunnable(engineJob,
decodeJob, priority); jobs.put(key, engineJob);
engineJob.addCallback(cb);
engineJob.start(runnable); if (Log.isLoggable(TAG,
Log.VERBOSE)) { logWithTimeAndKey("Started new
load", startTime, key); } return new
LoadStatus(cb, engineJob); } ...}
```

可以看到，这里在第17行调用了loadFromCache()方法来获取缓存图片，如果获取到就直接调用cb.onResourceReady()方法进行回调。如果没有获取到，则会在第26行调用loadFromActiveResources()方法来获取缓存图片，获取到的话也直接进行回调。只有在两个方法都没有获取到缓存的情况下，才会继续向下执行，从而开启线程来加载图片。

也就是说，Glide的图片加载过程中会调用两个方法来获取内存缓存，loadFromCache()和loadFromActiveResources()。这两个方法中一个使用的就是LruCache算法，另一个使用的就是弱引用。我们来看一下它们的源码：

```
public class Engine implements EngineJobListener,
MemoryCache.ResourceRemovedListener,
EngineResource.ResourceListener {    private final
MemoryCache cache;    private final Map<Key,
WeakReference<EngineResource<?>>> activeResources;    ...
    private EngineResource<?> loadFromCache(Key key,
boolean isMemoryCacheable) {        if
(!isMemoryCacheable) {            return null;        }
        EngineResource<?> cached =
getEngineResourceFromCache(key);        if (cached !=
null) {            cached.acquire();            activeResources.put(key, new ResourceWeakReference(key,
cached, getReferenceQueue()));        }        return
cached;    }    private EngineResource<?>
getEngineResourceFromCache(Key key) {        Resource<?>
cached = cache.remove(key);        final EngineResource
result;        if (cached == null) {            result =
null;        } else if (cached instanceof EngineResource) {
            result = (EngineResource) cached;        } else {
            result = new EngineResource(cached,
true /*isCacheable*/);        }        return result;
    }    private EngineResource<?>
loadFromActiveResources(Key key, boolean
isMemoryCacheable) {        if (!isMemoryCacheable) {
            return null;        }        EngineResource<?>
active = null;        WeakReference<EngineResource<?>>
activeRef = activeResources.get(key);        if
(activeRef != null) {            active =
activeRef.get();            if (active != null) {
                active.acquire();            } else {
                activeResources.remove(key);            }
        }        return active;    }    ...}
```

在loadFromCache()方法一开始，首先就判断了isMemoryCacheable是不是false，如果是false的话就直接返回null。这是什么意思呢？其实很简单，我们刚刚不是学了一个skipMemoryCache()方法吗？如果在这个方法中传入true，那么这里的isMemoryCacheable就会是false，表示内存

缓存已被禁用。

我们继续往下看，接着调用了getEngineResourceFromCache()方法来获取缓存。在这个方法中，会使用缓存Key来从cache当中取值，而这里的cache对象就是在构建Glide对象时创建的LruResourceCache，那么说明这里其实使用的就是LruCache算法了。

但是呢，观察第22行，当我们从LruResourceCache中获取到缓存图片之后会将它从缓存中移除，然后在第16行将这个缓存图片存储到activeResources当中。activeResources就是一个弱引用的HashMap，用来缓存正在使用中的图片，我们可以看到，loadFromActiveResources()方法就是从activeResources这个HashMap当中取值的。使用activeResources来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

好的，从内存缓存中读取数据的逻辑大概就是这些了。概括一下来说，就是如果能从内存缓存当中读取到要加载的图片，那么就直接进行回调，如果读取不到的话，才会开启线程执行后面的图片加载逻辑。

现在我们已经搞明白了内存缓存读取的原理，接下来的问题就是内存缓存是在哪里写入的呢？这里我们又要回顾一下上一篇文章中的内容了。还记得我们之前分析过，当图片加载完成之后，会在EngineJob当中通过Handler发送一条消息将执行逻辑切回到主线程当中，从而执行handleResultOnMainThread()方法。那么我们现在重新来看一下这个方法，代码如下所示：

```
class EngineJob implements  
EngineRunnable.EngineRunnableManager {    private final  
EngineResourceFactory engineResourceFactory;    ...  
private void handleResultOnMainThread() {        if  
(isCancelled) {            resource.recycle();  
return;        } else if (cbs.isEmpty()) {  
throw new IllegalStateException("Received a resource  
without any callbacks to notify");        }  
engineResource = engineResourceFactory.build(resource,  
isCacheable);        hasResource = true;  
engineResource.acquire();  
listener.onEngineJobComplete(key, engineResource);  
for (ResourceCallback cb : cbs) {            if  
(!isInIgnoredCallbacks(cb)) {  
engineResource.acquire();  
cb.onResourceReady(engineResource);            }        }  
        engineResource.release();    }    static class  
EngineResourceFactory {        public <R>  
EngineResource<R> build(Resource<R> resource, boolean  
isMemoryCacheable) {            return new  
EngineResource<R>(resource, isMemoryCacheable);        }  
    }    ...}
```

在第13行，这里通过EngineResourceFactory构建出了一个包含图片资源的EngineResource对象，然后会在第16行将这个对象回调到Engine的onEngineJobComplete()方法当中，如下所示：

```
public class Engine implements EngineJobListener,
MemoryCache.ResourceRemovedListener,
EngineResource.ResourceListener {    ...           @Override
    public void onEngineJobComplete(Key key,
EngineResource<?> resource) {
    util.assertMainThread();          // A null resource
    indicates that the load failed, usually due to an
    exception.          if (resource != null) {
    resource.setResourceListener(key, this);          if
    (resource.isCacheable()) {
    activeResources.put(key, new ResourceWeakReference(key,
    resource, getReferenceQueue()));          }
    jobs.remove(key);    }    ...}
```

现在就非常明显了，可以看到，在第13行，回调过来的EngineResource被put到了activeResources当中，也就是在这里写入的缓存。

那么这只是弱引用缓存，还有另外一种LruCache缓存是在哪里写入的呢？这就要介绍一下EngineResource中的一个引用机制了。观察刚才的handleResultOnMainThread()方法，在第15行和第19行有调用EngineResource的acquire()方法，在第23行有调用它的release()方法。其实，EngineResource是用一个acquired变量用来记录图片被引用的次数，调用acquire()方法会让变量加1，调用release()方法会让变量减1，代码如下所示：

```
class EngineResource<Z> implements Resource<Z> {  
    private int acquired;      ...      void acquire() {  
        if (isRecycled) {           throw new  
IllegalStateException("Cannot acquire a recycled  
resource"); }           if  
(!Looper.getMainLooper().equals(Looper.myLooper())) {  
            throw new IllegalThreadStateException("Must call  
acquire on the main thread"); }           ++acquired;  
    }    void release() {           if (acquired <= 0) {  
            throw new IllegalStateException("Cannot release a  
recycled or not yet acquired resource"); }  
    if (!Looper.getMainLooper().equals(Looper.myLooper())) {  
        throw new IllegalThreadStateException("Must  
call release on the main thread"); }           if (--  
acquired == 0) {  
        listener.onResourceReleased(key, this); }     }}}
```

也就是说，当acquired变量大于0的时候，说明图片正在使用中，也就应该放到activeResources弱引用缓存当中。而经过release()之后，如果acquired变量等于0了，说明图片已经不再被使用了，那么此时会在第24行调用listener的onResourceReleased()方法来释放资源，这个listener就是Engine对象，我们来看下它的onResourceReleased()方法：

```
public class Engine implements EngineJobListener,  
MemoryCache.ResourceRemovedListener,  
EngineResource.ResourceListener {    private final  
MemoryCache cache;    private final Map<Key,  
WeakReference<EngineResource<?>>> activeResources;    ...  
    @Override    public void onResourceReleased(Key  
cacheKey, EngineResource resource) {  
        util.assertMainThread();  
        activeResources.remove(cacheKey);           if  
(resource.isCacheable()) {           cache.put(cacheKey,  
resource); } else {  
        resourceRecycler.recycle(resource); }     }    ...}
```

可以看到，这里首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

这就是Glide内存缓存的实现原理。

总结：

内存缓存读取：先获取LruResourceCache（Lru算法缓存），然后放入activeResources（是一个弱引用的HashMap）来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

内存缓存写入：回调过来的EngineResource先被put到了activeResources当中，也就是在这里写入的缓存。如果图片正在使用中，也就应该放到activeResources弱引用缓存当中。如果图片不再被使用了，首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

进一步总结：正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

磁盘缓存：

接下来我们开始学习硬盘缓存方面的内容。

主要作用就是防止重复将图片读取到我们的磁盘当中

磁盘缓存读取：

不知道你还记不记得，在本系列的第一篇文章中我们就使用过硬盘缓存的功能了。当时为了禁止Glide对图片进行硬盘缓存而使用了如下代码：

```
Glide.with(this)      .load(url)
    .diskCacheStrategy(DiskCacheStrategy.NONE)
    .into(imageview);
```

调用diskCacheStrategy()方法并传入DiskCacheStrategy.NONE，就可以禁用掉Glide的硬盘缓存功能了。

这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收四种参数：

- DiskCacheStrategy.NONE： 表示不缓存任何内容。
- DiskCacheStrategy.SOURCE： 表示只缓存原始图片。
- DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。
- DiskCacheStrategy.ALL： 表示既缓存原始图片，也缓存转换过后的图片。

上面四种参数的解释本身并没有什么难理解的地方，但是有一个概念大家需要了解，就是当我们使用Glide去加载一张图片的时候，Glide默认并不会将原始图片展示出来，而是会对图片进行压缩和转换（我们会在后面学习这方面的内容）。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片。而Glide默认情况下在硬盘缓存的就是转换过后的图片，我们通过调用diskCacheStrategy()方法则可以改变这一默认行为。

好的，关于Glide硬盘缓存的用法也就只有这么多，那么接下来还是老套路，我们通过阅读源码来分析一下，Glide的硬盘缓存功能是如何实现的。

首先，和内存缓存类似，硬盘缓存的实现也是使用的LruCache算法，而且Google还提供了一个现成的工具类DiskLruCache。我之前也专门写过一篇文章对这个DiskLruCache工具进行了比较全面的分析，感兴趣的朋友可以参考一下Android DiskLruCache完全解析，硬盘缓存的最佳方案。当然，Glide是使用的自己编写的DiskLruCache工具类，但是基本的实现原理都是差不多的。

接下来我们看一下Glide是在哪里读取硬盘缓存的。这里又需要回忆一下上篇文章中的内容了，Glide开启线程来加载图片后会执行EngineRunnable的run()方法，run()方法中又会调用一个decode()方法，那么我们重新再来看一下这个decode()方法的源码：

```
private Resource<?> decode() throws Exception {    if  
    (isDecodingFromCache()) {        return  
    decodeFromCache();    } else {        return  
    decodeFromSource();    }}
```

可以看到，这里会分为两种情况，一种是调用decodeFromCache()方法从硬盘缓存当中读取图片，一种是调用decodeFromSource()来读取原始图片。默认情况下Glide会优先从缓存当中读取，只有缓存中不存在要读取的图片时，才会去读取原始图片。那么我们现在来看一下decodeFromCache()方法的源码，如下所示：

```
private Resource<?> decodeFromCache() throws Exception {  
    Resource<?> result = null; try { result =  
decodeJob.decodeResultFromCache(); } catch (Exception  
e) { if (Log.isLoggable(TAG, Log.DEBUG)) {  
    Log.d(TAG, "Exception decoding result from cache: " +  
e); } if (result == null) { result =  
decodeJob.decodeSourceFromCache(); } return  
result;}
```

可以看到，这里会先去调用DecodeJob的decodeResultFromCache()方法来获取缓存，如果获取不到，会再调用decodeSourceFromCache()方法获取缓存，这两个方法的区别其实就是DiskCacheStrategy.RESULT和DiskCacheStrategy.SOURCE这两个参数的区别，相信不需要我再做什么解释吧。

那么我们来看一下这两个方法的源码吧，如下所示：

```
public Resource<Z> decodeResultFromCache() throws  
Exception { if (!diskCacheStrategy.cacheResult()) {  
    return null; } long startTime =  
LogTime.getLogTime(); Resource<T> transformed =  
loadFromCache(resultKey); startTime =  
LogTime.getLogTime(); Resource<Z> result =  
transcode(transformed); return result;}public  
Resource<Z> decodeSourceFromCache() throws Exception {  
if (!diskCacheStrategy.cacheSource()) { return  
null; } long startTime = LogTime.getLogTime();  
Resource<T> decoded =  
loadFromCache(resultKey.getOriginalKey()); return  
transformEncodeAndTranscode(decoded);}
```

可以看到，它们都是调用了loadFromCache()方法从缓存当中读取数据，如果是decodeResultFromCache()方法就直接将数据解码并返回，如果是decodeSourceFromCache()方法，还要调用一下transformEncodeAndTranscode()方法先将数据转换一下再解码并返回。

然而我们注意到，这两个方法中在调用loadFromCache()方法时传入的参数却不一样，一个传入的是resultKey，另外一个却又调用了resultKey的getOriginalKey()方法。这个其实非常好理解，刚才我们已经解释过了，Glide的缓存Key是由10个参数共同组成的，包括图片的width、height等等。但如果我们是缓存的原始图片，其实并不需要这么多的参数，因为不用对图片做任何的变化。那么我们来看一下getOriginalKey()方法的源码：

```
public Key getOriginalKey() {    if (originalKey == null) {        originalKey = new OriginalKey(id, signature);    }    return originalKey;}
```

可以看到，这里其实就是忽略了绝大部分的参数，只使用了id和signature这两个参数来构成缓存Key。而signature参数绝大多数情况下都是用不到的，因此基本上可以说就是由id（也就是图片url）来决定的Original缓存Key。

搞明白了这两种缓存Key的区别，那么接下来我们看一下loadFromCache()方法的源码吧：

```
private Resource<T> loadFromCache(Key key) throws IOException {    File cacheFile = diskCacheProvider.getDiskCache().get(key);    if (cacheFile == null) {        return null;    }    Resource<T> result = null;    try {        result = loadProvider.getCacheDecoder().decode(cacheFile, width, height);    } finally {        if (result == null) {            diskCacheProvider.getDiskCache().delete(key);        }    }    return result;}
```

这个方法的逻辑非常简单，调用getDiskCache()方法获取到的就是Glide自己编写的DiskLruCache工具类的实例，然后调用它的get()方法并把缓存Key传入，就能得到硬盘缓存的文件了。如果文件为空就返回null，如果文件不为空则将它解码成Resource对象后返回即可。

这样我们就将硬盘缓存读取的源码分析完了，那么硬盘缓存又是在哪里写入的呢？趁热打铁我们赶快继续分析下去。

磁盘缓存写入：

刚才已经分析过了，在没有缓存的情况下，会调用decodeFromSource()方法来读取原始图片。那么我们来看下这个方法：

```
public Resource<Z> decodeFromSource() throws Exception {  
    Resource<T> decoded = decodeSource();      return  
    transformEncodeAndTranscode(decoded);}
```

这个方法中只有两行代码，decodeSource()顾名思义是用来解析原图片的，而transformEncodeAndTranscode()则是用来对图片进行转换和转码的。我们先来看decodeSource()方法：

```
private Resource<T> decodeSource() throws Exception {
    Resource<T> decoded = null; try { long
        startTime = LogTime.getLogTime(); final A data =
        fetcher.loadData(priority); if (isCancelled) {
            return null; } decoded =
        decodeFromSourceData(data); } finally {
        fetcher.cleanup(); } return decoded;}private
    Resource<T> decodeFromSourceData(A data) throws
    IOException { final Resource<T> decoded; if
    (diskCacheStrategy.cacheSource()) { decoded =
    cacheAndDecodeSourceData(data); } else { long
        startTime = LogTime.getLogTime(); decoded =
    loadProvider.getSourceDecoder().decode(data, width,
    height); } return decoded;}private Resource<T>
    cacheAndDecodeSourceData(A data) throws IOException {
    long startTime = LogTime.getLogTime(); SourceWriter<A>
    writer = new SourceWriter<A>
    (loadProvider.getSourceEncoder(), data);
    diskCacheProvider.getDiskCache().put(resultKey.getOriginal
    Key(), writer); startTime = LogTime.getLogTime();
    Resource<T> result =
    loadFromCache(resultKey.getOriginalKey()); return
    result;}
```

这里会在第5行先调用fetcher的loadData()方法读取图片数据，然后在第9行调用decodeFromSourceData()方法来对图片进行解码。接下来会在第18行先判断是否允许缓存原始图片，如果允许的话又会调用cacheAndDecodeSourceData()方法。而在这个方法中同样调用了getDiskCache()方法来获取DiskLruCache实例，接着调用它的put()方法就可以写入硬盘缓存了，注意原始图片的缓存Key是用的resultKey.getOriginalKey()。

好的，原始图片的缓存写入就是这么简单，接下来我们分析一下transformEncodeAndTranscode()方法的源码，来看看转换过后的图片缓存是怎么写入的。代码如下所示：

```
private Resource<Z>
transformEncodeAndTranscode(Resource<T> decoded) {
    long startTime = LogTime.getLogTime();      Resource<T>
    transformed = transform(decoded);
    writeTransformedToCache(transformed);      startTime =
    LogTime.getLogTime();      Resource<Z> result =
    transcode(transformed);      return result;}private void
writeTransformedToCache(Resource<T> transformed) {      if
(transformed == null || !diskCacheStrategy.cacheResult())
{          return;      }      long startTime =
    LogTime.getLogTime();      SourceWriter<Resource<T>> writer
= new SourceWriter<Resource<T>>
    (loadProvider.getEncoder(), transformed);
    diskCacheProvider.getDiskCache().put(resultKey, writer);}
```

这里的逻辑就更加简单明了了。先是在第3行调用transform()方法来对图片进行转换，然后在writeTransformedToCache()方法中将转换过后的图片写入到硬盘缓存中，调用的同样是DiskLruCache实例的put()方法，不过这里用的缓存Key是resultKey。

这样我们就将Glide硬盘缓存的实现原理也分析完了。虽然这些源码看上去如此的复杂，但是经过Glide出色的封装，使得我们只需要通过skipMemoryCache()和diskCacheStrategy()这两个方法就可以轻松自如地控制Glide的缓存功能了。

了解了Glide缓存的实现原理之后，接下来我们再来学习一些Glide缓存的高级技巧吧。

高级技巧

虽说Glide将缓存功能高度封装之后，使得用法变得非常简单，但同时也带来了一些问题。

比如之前有一位群里的朋友就跟我说过，他们项目的图片资源都是存放在七牛云上面的，而七牛云为了对图片资源进行保护，会在图片url地址的基础之上再加上一个token参数。也就是说，一张图片的url地址可能会是如下格式：

```
http://url.com/image.jpg?token=d9caa6e02c990b0a
```

而使用Glide加载这张图片的话，也就会使用这个url地址来组成缓存Key。

但是接下来问题就来了，token作为一个验证身份的参数并不是一成不变的，很有可能时时刻刻都在变化。而如果token变了，那么图片的url也就跟着变了，图片url变了，缓存Key也就跟着变了。结果就造成了，明明是同一张图片，就因为token不断在改变，导致Glide的缓存功能完全失效了。

这其实是个挺棘手的问题，而且我相信绝对不仅仅是七牛云这一个个例，大家在使用Glide的时候很有可能都会遇到这个问题。

那么该如何解决这个问题呢？我们还是从源码的层面进行分析，首先再来看一下Glide生成缓存Key这部分的代码：

```
public class Engine implements EngineJobListener,  
MemoryCache.ResourceRemovedListener,  
EngineResource.ResourceListener {    public <T, Z, R>  
LoadStatus load(Key signature, int width, int height,  
DataFetcher<T> fetcher,                      DataLoadProvider<T, Z>  
loadProvider, Transformation<Z> transformation,  
ResourceTranscoder<Z, R> transcoder,          Priority  
priority, boolean isMemoryCacheable, DiskCacheStrategy  
diskCacheStrategy, ResourceCallback cb) {  
    Util.assertMainThread();        long startTime =  
    LogTime.getLogTime();        final String id =  
    fetcher.getId();            EngineKey key =  
    keyFactory.buildKey(id, signature, width, height,  
    loadProvider.getCacheDecoder(),  
    loadProvider.getSourceDecoder(), transformation,  
    loadProvider.getEncoder(),                  transcoder,  
    loadProvider.getSourceEncoder());           ...     ...}
```

来看一下第11行，刚才已经说过了，这个id其实就是图片的url地址。那么，这里是通过调用fetcher.getId()方法来获取的图片url地址，而我们在上一篇文章中已经知道了，fetcher就是HttpUrlFetcher的实例，我们就来看一下它的getId()方法的源码吧，如下所示：

```
public class HttpUrlFetcher implements  
DataFetcher<InputStream> {    private final GlideUrl  
glideUrl;    ...    public HttpUrlFetcher(GlideUrl  
glideUrl) {        this(glideUrl,  
DEFAULT_CONNECTION_FACTORY);    }  
HttpUrlFetcher(GlideUrl glideUrl,  
HttpUrlConnectionFactory connectionFactory) {  
this.glideUrl = glideUrl;        this.connectionFactory =  
connectionFactory;    }    @Override    public String  
getId() {        return glideUrl.getCacheKey();    }  
...}
```

可以看到，getId()方法中又调用了GlideUrl的getCacheKey()方法。那么这个GlideUrl对象是从哪里来的呢？其实就是我们在load()方法中传入的图片url地址，然后Glide在内部把这个url地址包装成了一个GlideUrl对象。

很明显，接下来我们就要看一下GlideUrl的getCacheKey()方法的源码了，如下所示：

```
public class GlideUrl {    private final URL url;  
private final String urlString;    ...    public  
GlideUrl(URL url) {        this(url, Headers.DEFAULT);  
}    public GlideUrl(String url) {        this(url,  
Headers.DEFAULT);    }    public GlideUrl(URL url,  
Headers headers) {        ...        this.url = url;  
        urlString = null;    }    public GlideUrl(String url,  
Headers headers) {        ...        this.stringUrl =  
url;        this.url = null;    }    public String  
getCacheKey() {        return urlString != null ?  
stringUrl : url.toString();    }    ...}
```

这里我将代码稍微进行了一点简化，这样看上去更加简单明了。GlideUrl类的构造函数接收两种类型的参数，一种是url字符串，一种是URL对象。然后getCacheKey()方法中的判断逻辑非常简单，如果传入的是url字符串，那么就直接返回这个字符串本身，如果传入的是URL对象，那么就返回这个对象toString()后的结果。

其实看到这里，我相信大家已经猜到解决方案了，因为getCacheKey()方法中的逻辑太直白了，直接就是将图片的url地址进行返回来作为缓存Key的。那么其实我们只需要重写这个getCacheKey()方法，加入一些自己的逻辑判断，就能轻松解决掉刚才的问题了。

创建一个MyGlideUrl继承自GlideUrl，代码如下所示：

```
public class MyGlideUrl extends GlideUrl {    private
String murl;    public MyGlideUrl(String url) {
super(url);        murl = url;    }    @Override
public String getCacheKey() {        return
murl.replace(findTokenParam(), "");    }    private
String findTokenParam() {        String tokenParam = "";
int tokenKeyIndex = murl.indexOf("?token=") >= 0 ?
murl.indexOf("?token=") : murl.indexOf("&token=");
if (tokenKeyIndex != -1) {            int nextAndIndex =
murl.indexOf("&", tokenKeyIndex + 1);            if
(nextAndIndex != -1) {                tokenParam =
murl.substring(tokenKeyIndex + 1, nextAndIndex + 1);
} else {                tokenParam =
murl.substring(tokenKeyIndex);            }
}
return tokenParam;    }}
```

可以看到，这里我们重写了getCacheKey()方法，在里面加入了一段逻辑用于将图片url地址中token参数的这一部分移除掉。这样getCacheKey()方法得到的就是一个没有token参数的url地址，从而不管token怎么变化，最终Glide的缓存Key都是固定不变的了。

当然，定义好了MyGlideUrl，我们还得使用它才行，将加载图片的代码改成如下方式即可：

```
Glide.with(this)      .load(new MyGlideurl(url))  
.into(imageview);
```

也就是说，我们需要在load()方法中传入这个自定义的MyGlideUrl对象，而不能再像之前那样直接传入url字符串了。不然的话Glide在内部还是会使用原始的GlideUrl类，而不是我们自定义的MyGlideUrl类。

这样我们就将这个棘手的缓存问题给解决掉了。

第四节 day 20 面试题：Glide面试题

1. Glide的优点
2. 生命周期绑定原理
3. 缓存原理

4.1 Glide的优点

1. 使用简单，链式调用比较方便

```
Glide.with(context)      .load(uri)  
.into(imageview);
```

2. 占用内存较小

默认使用RGB_565格式，是Picasso的内存占用的一半（Picasso使用RGB_8888）

3. 无代码侵入

相对Picasso来说，接入很简单，无需将ImageView替代为自定义View，也不需要其他配置，只需要将库引入即可

4. 支持gif

ImageLoader不支持gif图片加载

5. 缓存优化

1. 支持内存分级缓存：正在使用的图片，弱引用缓存；已使用过的图片LruCache缓存
2. Glide会为不同的ImageView尺寸缓存不同尺寸的图片
6. 与Activity生命周期绑定，不会出现内存泄露

4.2 生命周期绑定原理

1. 实现原理

基于在Activity中添加无UI的Fragment，通过Fragment接收Activity传递的生命周期。Fragment和RequestManager基于LifeCycle接口建立联系，并传递生命周期事件，实现生命周期感知。

2. 如何绑定生命周期

在调用Glide.with(Activity activity)的时候，我们跟一下流程，核心代码下面看一下。

```
// with入口 public static RequestManager with(@NotNull  
FragmentActivity activity) { return  
getRetriever(activity).get(activity); } // 此处拿到对应的  
FragmentManager, 为生成Fragment做准备 public  
RequestManager get(@NotNull FragmentActivity activity)  
{ if(Util.isOnBackgroundThread()) { return  
this.get(activity.getApplicationContext()); } else  
{ assertNotDestroyed(activity);  
android.support.v4.app.FragmentManager fm =  
activity.getSupportFragmentManager(); return  
this.supportFragmentGet(activity, fm, (Fragment)null,  
isActivityVisible(activity)); } } private  
RequestManager supportFragmentGet(@NotNull Context  
context, @NotNull  
android.support.v4.app.FragmentManager fm, @Nullable  
Fragment parentHint, boolean isVisible) { //  
current就是一个无UI的Fragment实例  
SupportRequestManagerFragment current =  
this.getSupportRequestManagerFragment(fm, parentHint,  
isVisible); RequestManager requestManager =  
current.getRequestManager(); if(requestManager ==  
null) { Glide glide = Glide.get(context);  
// 将Fragment的Lifecycle传入RequestManager中, 建立起来联系  
requestManager = this.factory.build(glide,  
current.getGlideLifecycle(),  
current.getRequestManagerTreeNode(), context);  
current.setRequestManager(requestManager); }  
return requestManager; } //RequestManager的构造方法中绑定  
Lifecycle, 将自己的引用存入Lifecycle, 调用Lifecycle的生命周期  
时进行回调 lifecycle.addListener(this);
```

稍微整理一下就是：

1. Glide绑定Activity时，生成一个无UI的Fragment
2. 将无UI的Fragment的Lifecycle传入到RequestManager中
3. 在RequestManager的构造方法中，将RequestManager存入到之前传入的Fragment的Lifecycle，在回调Lifecycle时会回调到Glide的相应方法

3. 如何通过Fragment的生命周期回调调用Glide的对应方法

通过Fragment的回调调用到Glide的RequestManager的对应的方法即可执行不同的操作，主要绑定的三个方法为：

`onStart()`,`onStop()`,`onDestroy()`

回调的源码也翻一下

```
//RequestManager的构造方法中绑定Lifecycle, 将自己的引用存入
Lifecycle, 调用Lifecycle的生命周期时进行回调 //这个this是
RequestManager的实例 lifecycle.addListener(this); //
onDestory的回调示例 void onDestroy() {
this.isDestroyed = true;      Iterator var1 =
util.getSnapshot(this.lifecycleListeners).iterator();
while(var1.hasNext()) {      LifecycleListener
lifecycleListener = (LifecycleListener)var1.next();
lifecycleListener.onDestroy(); } } //下面看一下
RequestManager里面的onDestory方法, 里面主要做一些解绑和清除操
作 public void onDestroy() {
this.targetTracker.onDestroy();      Iterator var1 =
this.targetTracker.getAll().iterator();
while(var1.hasNext()) {      Target<?> target =
(Target)var1.next();      this.clear(target);
this.targetTracker.clear();
this.requestTracker.clearRequests();
this.lifecycle.removeListener(this);
this.lifecycle.removeListener(this.connectivityMonitor)
;
this.mainHandler.removeCallbacks(this.addSelfToLifecycle);
      this.glide.unregisterRequestManager(this); }}
```

4.3 缓存原理

1. 几种缓存模式

以下几个属性是4.7.0版本的

- `DiskCacheStrategy.ALL`: 原始图片和转换过的图片都缓存
- `DiskCacheStrategy.RESOURCE`: 只缓存原始图片

- DiskCacheStrategy.NONE：不缓存
- DiskCacheStrategy.DATA：只缓存使用过的图片

2. 内存缓存

这部分逻辑较多，记住结论就行，正在使用的图片使用的是弱引用缓存，使用完成后，添加到LruCache缓存

3. 磁盘缓存

默认缓存使用过的分辨率图片，使用DiskLruCache来做的磁盘缓存

主要就是根据配置的策略去读取对应的缓存，原始数据缓存是在数据请求完成时，转换过的图片是在对图片进行transform转换后缓存。

第五节 聊一聊关于Glide在面试中的那些事

5.1 前言

今天填完离职表，帮着公司面试几个帮着填坑的同行，聊着聊着就聊到了Glide，信誓旦旦的和我说，这块很熟悉，之前在掘金社区里，看过一个叫蓝师傅写的这块的文章，基本细节都掌握@@。 (一副胜利在望的表情，好吧，正巧我也看过，咱们就问一问看的仔细情况)

5.2 来简单介绍下Glide的缓存

5.2.1 分析

(这货滔滔不绝的说了一大通，从LruCache说到了LinkedHashMap，巴拉巴拉。。。个人建议这块一定要简述，面试时原理说的太多，第一很多细节会被打断问到，第二点，说这么多，给人的感觉就是在背诵东西，原理概括能力很弱或者感觉根本就没有自己的体会。)

5.2.2 答案

Glide的缓存机制，主要分为2种缓存，一种是内存缓存，一种是磁盘缓存。

之所以使用内存缓存的原因是：防止应用重复将图片读入到内存，造成内存资源浪费。

之所以使用磁盘缓存的原因是：防止应用重复的从网络或者其他地方下载和读取数据。

正式因为有着这两种缓存的结合，才构成了Glide极佳的缓存效果。

(先告诉人家有哪几种缓存，主要是为了什么目的才用的缓存，然后可以看着面试官，要么等着他继续问，如果他不问，等着你，这个时候你就可以继续的往细节处介绍

5.3 嗯，具体说一说Glide的三级缓存原理

5.3.1 分析

(记得，如果需要具体谈原理时，要先宏观，后细节)

5.3.2 答案

读取一张图片的时候，获取顺序：

Lru算法缓存-》弱引用缓存-》磁盘缓存（如果设置了的话）

当我们的APP中想要加载某张图片时，先去LruCache中寻找图片，如果LruCache中有，则直接取出来使用，并将该图片放入WeakReference中，如果LruCache中没有，则去WeakReference中寻找，如果WeakReference中有，则从WeakReference中取出图片使用，如果WeakReference中也没有图片，则从磁盘缓存/网络中加载图片。

注：图片正在使用时存在于 activeResources 弱引用map中

流程如下图

将图片缓存的时候，写入顺序：

弱引用缓存-》Lru算法缓存-》磁盘缓存中

当图片不存在的时候，先从网络下载图片，然后将图片存入弱引用中，glide会采用一个acquired (int) 变量用来记录图片被引用的次数，

当acquired变量大于0的时候，说明图片正在使用中，也就是将图片放到弱引用缓存当中；

如果acquired变量等于0了，说明图片已经不再被使用了，那么此时会

调用方法来释放资源，首先会将缓存图片从弱引用中移除，然后再将它put到LruResourceCache当中。

这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

引深：

关于LruCache

最近最少使用算法，设定一个缓存大小，当缓存达到这个大小之后，会将最老的数据移除，避免图片占用内存过大导致OOM。

LruCache 内部用LinkHashMap存取数据，在双向链表保证数据新旧顺序的前提下，设置一个最大内存，往里面put数据的时候，当数据达到最大内存的时候，将最老的数据移除掉，保证内存不超过设定的最大值。

关于LinkedHashMap

LinkHashMap 继承HashMap，在HashMap的基础上，新增了双向链表结构，每次访问数据的时候，会更新被访问的数据的链表指针，具体就是先在链表中删除该节点，然后添加到链表头header之前，这样就保证了链表头header节点之前的数据都是最近访问的（从链表中删除并不是真的删除数据，只是移动链表指针，数据本身在map中的位置是不变的）

5.4.Glide加载一个一兆的图片（100 * 100），是否会压缩后再加载，放到一个300 * 300的view上会怎样，800*800呢，图片会很模糊，怎么处理？

5.4.1 分析

（因为你缓存机制无论是看博客还是看一些面试宝典，如果只是考原理或者定义，光把上面的文字背诵下来就可以了，但是背诵和真正的理解是两回事，自己没有形成感悟，不理解这个框架，只是一味的迎合面试，这个问题就可以卡住你，另外千万别和面试官嘚瑟，果然，这个面试的哥们，这块就卡住了，支支吾吾的半天没答上来，果然是只看了博客，没真正的阅读过源码）

5.4.2 答案

当我们调整imageview的大小时，Picasso会不管imageview大小是什么，总是直接缓存整张图片，而Glide就不一样了，它会为每个不同尺寸的Imageview缓存一张图片，也就是说不管你的这张图片有没有加载过，只要imageview的尺寸不一样，那么Glide就会重新加载一次，这时候，它会在加载的imageview之前从网络上重新下载，然后再缓存。

举个例子，如果一个页面的imageview是300 * 300像素，而另一个页面中的imageview是100 * 100像素，这时候想要让两个imageview像是同一张图片，那么Glide需要下载两次图片，并且缓存两张图片。

```
public <R> LoadStatus load() {    // 根据请求参数得到缓存的键  
    EngineKey key = keyFactory.buildKey(model, signature,  
    width, height, transformations, resourceClass,  
    transcodeClass, options);}
```

看到了吧，缓存Key的生成条件之一就是控件的长宽。

5.5 简单说一下内存泄漏的场景，如果在一个页面中使用Glide加载了一张图片，图片正在获取中，如果突然关闭页面，这个页面会造成内存泄漏吗？

5.5.1 分析

(注意一定要审题，因为之前问了这个小伙，内存泄漏的原因，无非是长生命周期引用了短生命周期的对象等等，然后突然画风一变，直接问了Glide加载图片会不会引起图片泄漏，这个小伙想也没想，直接回答道会引起内存泄漏，可以用LeakCanary检测，巴拉巴拉。。。)

5.5.2 答案

因为Glide在加载资源的时候，如果是在Activity、Fragment这一类有生命周期的组件上进行的话，会创建一个透明的RequestManagerFragment加入到FragmentManager之中，感知生命周期，当Activity、Fragment等组件进入不可见，或者已经销毁的时候，Glide会停止加载资源。

但是如果，是在非生命周期的组件上进行时，会采用Application 的生命周期贯穿整个应用，所以 applicationManager 只有在应用程序关闭的时候终止加载。

5.6如何设计一个大图加载框架

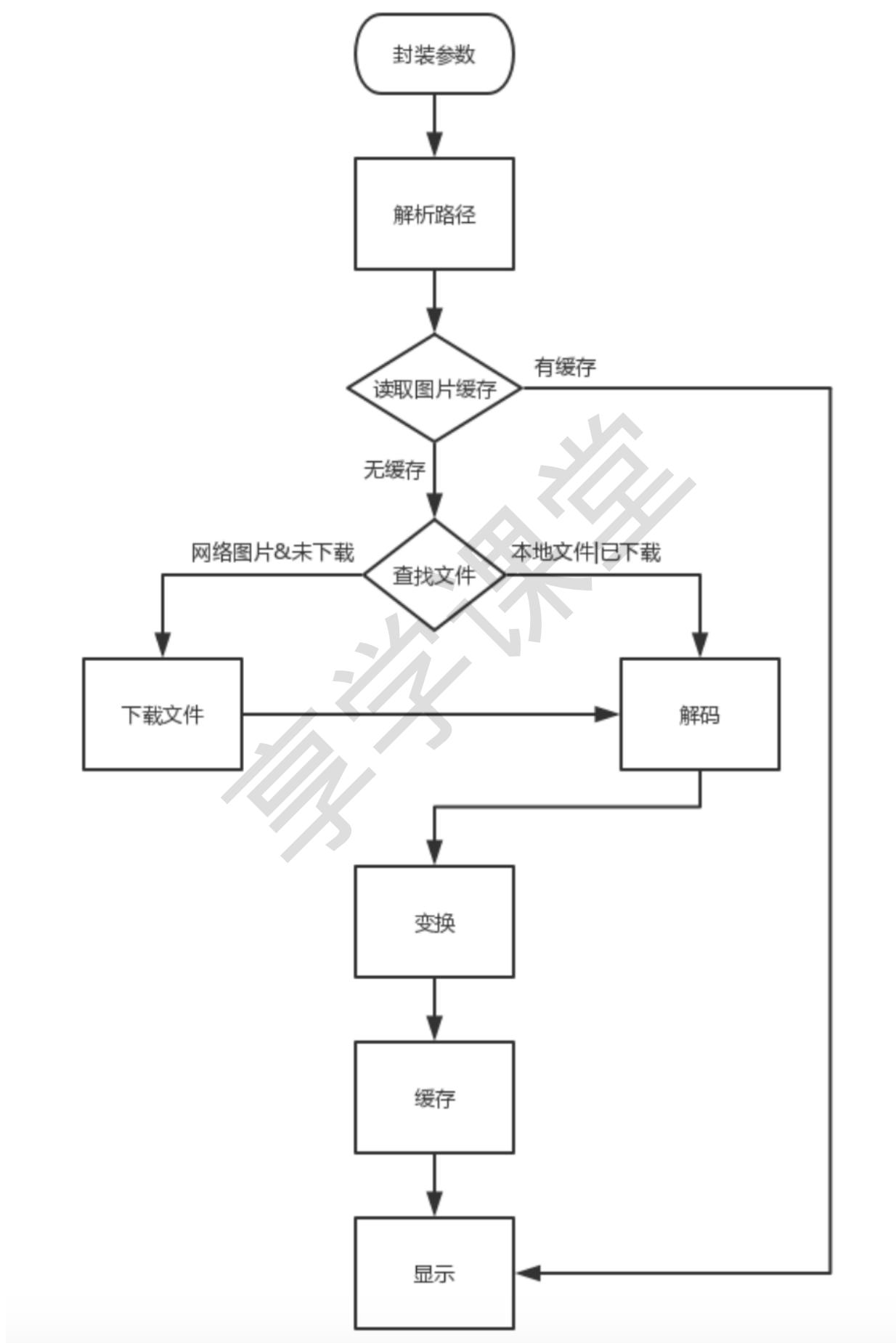
5.6.1 分析

(这个孩子，总算是羞愧的低下了头，一脸懵逼的和我说，这个我忘记了)

5.6.2 答案



概括来说，图片加载包含封装，解析，下载，解码，变换，缓存，显示等操作。



1. 封装参数：从指定来源，到输出结果，中间可能经历很多流程，所以第一件事就是封装参数，这些参数会贯穿整个过程；
2. 解析路径：图片的来源有多种，格式也不尽相同，需要规范化；
3. 读取缓存：为了减少计算，通常都会做缓存；同样的请求，从缓存中取图片（Bitmap）即可；
4. 查找文件/下载文件：如果是本地的文件，直接解码即可；如果是网络图片，需要先下载；
5. 解码：这一步是整个过程中最复杂的步骤之一，有不少细节，下个博客会说；
6. 变换：解码出Bitmap之后，可能还需要做一些变换处理（圆角，滤镜等）；
7. 缓存：得到最终bitmap之后，可以缓存起来，以便下次请求时直接取结果；
8. 显示：显示结果，可能需要做些动画（淡入动画，crossFade等）。

5.6.3END

先写到这，基本上问的也就这么多了，要是再细扣，感觉有些故意为难了，面试的一般都是在30分钟以内，感觉对方会的，我会跳过，因为你会了，我就没必要问了，浪费时间，感觉不会的，问道你说出不会或者不清楚为止，我也不可能再在这方面细扣，原因也是一样，浪费时间。主要是发现每个人的长处，及对待事情的态度。

另外其实吧，不一定都答的上来才会要你，也不是因为答不上来就不要你，主要吧，还是靠感觉，好相处，人又不是太滑，性格nice，给人踏实的感觉的，都会网开一面的。

下期我会仿照着Glide自己仿照一个，将Glide的主流程所涉及的，都会加上，这样方便大家的理解。有兴趣的可以关注我。

还是那句话

虽然更新时间没准，快则半天，慢则半年，但是每一篇文章，我都深入浅出，要么丰富的流程图，一看就懂，要么就是一步一步仿照着写，学习框架原理还是那句话，先搞清楚为什么要这么设计，可不可以不这么写或者不加这个功能，根据情境分析，然后再去解读这块源码逻辑，这样才会记的牢固

第六节 面试官：简历上如果写Glide，请注意以下几点...

这次来面试的是一个有着5年工作经验的小伙，截取了一段对话如下：

面试官：我看你写到Glide，为什么用Glide，而不选择其它图片加载框架？

小伙：Glide 使用简单，链式调用，很方便，一直用这个。

面试官：有看过它的源码吗？跟其它图片框架相比有哪些优势？

小伙：没有，只是在项目中使用而已~

面试官：假如现在不让你用开源库，需要你自己写一个图片加载框架，你会考虑哪些方面的问题，说说大概的思路。

小伙：额~，压缩吧。

面试官：还有吗？

小伙：额~，这个没写过。

说到图片加载框架，大家最熟悉的莫过于Glide了，但我却不推荐简历上写熟悉Glide，除非你熟读它的源码，或者参与Glide的开发和维护。

在一般面试中，遇到图片加载问题的频率一般不会太低，只是问法会有一些差异，例如：

- 简历上写Glide，那么会问一下Glide的设计，以及跟其它同类框架的对比；
- 假如让你写一个图片加载框架，说说思路；
- 给一个图片加载的场景，比如网络加载一张或多张大图，你会怎么做；

带着问题进入正文~

6.1 谈谈Glide

6.1.1 Glide 使用有多简单？

Glide由于其口碑好，很多开发者直接在项目中使用，使用方法相当简单

github.com/bumptech/glide...

1、添加依赖：

```
implementation  
'com.github.bumptech.glide:glide:4.10.0' annotationProcess  
or 'com.github.bumptech.glide:compiler:4.10.0'
```

2、添加网络权限

```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

3、一句代码加载图片到ImageView

```
Glide.with(this).load(imgUrl).into(mIv1);
```

进阶一点的用法，参数设置

```
RequestOptions options = new RequestOptions()  
.placeholder(R.drawable.ic_launcher_background)  
.error(R.mipmap.ic_launcher)  
.diskCacheStrategy(DiskCacheStrategy.NONE)  
.override(200, 100);Glide.with(this)  
.load(imgUrl) .apply(options)  
.into(mIv2);
```

使用Glide加载图片如此简单，这让很多开发者省下自己处理图片的时间，图片加载工作全部交给Glide来就完事，同时，很容易就把图片处理的相关知识点忘掉。

6.1.2 为什么用Glide？

从前段时间面试的情况，我发现了这个现象：简历上写熟悉Glide的，基本都是熟悉使用方法，很多3年-6年工作经验，除了说Glide使用方便，不清楚Glide跟其他图片框架如Fresco的对比有哪些优缺点。

首先，当下流行的图片加载框架有那么几个，可以拿 Glide 跟[Fresco](#)对比，例如这些：

Glide：

- 多种图片格式的缓存，适用于更多的内容表现形式（如Gif、WebP、缩略图、Video）
- 生命周期集成（根据Activity或者Fragment的生命周期管理图片加载请求）
- 高效处理Bitmap（bitmap的复用和主动回收，减少系统回收压力）
- 高效的缓存策略，灵活（Picasso只会缓存原始尺寸的图片，Glide缓存的是多种规格），加载速度快且内存开销小（默认Bitmap格式的不同，使得内存开销是Picasso的一半）

Fresco：

- 最大的优势在于5.0以下(最低2.3)的bitmap加载。在5.0以下系统，Fresco将图片放到一个特别的内存区域(Ashmemp区)
- 大大减少OOM（在更底层的Native层对OOM进行处理，图片将不再占用App的内存）
- 适用于需要高性能加载大量图片的场景

对于一般App来说，Glide完全够用，而对于图片需求比较大的App，为了防止加载大量图片导致OOM，Fresco会更合适一些。并不是说用Glide会导致OOM，Glide默认用的内存缓存是LruCache，内存不会一直往上涨。

6.2假如让你自己写个图片加载框架，你会考虑哪些问题？

首先，梳理一下必要的图片加载框架的需求：

- 异步加载：线程池
- 切换线程：Handler，没有争议吧
- 缓存：LruCache、DiskLruCache
- 防止OOM：软引用、LruCache、图片压缩、Bitmap像素存储位置
- 内存泄露：注意ImageView的正确引用，生命周期管理
- 列表滑动加载的问题：加载错乱、队满任务过多问题

当然，还有一些不是必要的需求，例如加载动画等。

6.2.1 异步加载：

线程池，多少个？

缓存一般有三级，内存缓存、硬盘、网络。

由于网络会阻塞，所以读内存和硬盘可以放在一个线程池，网络需要另外
一个线程池，网络也可以采用Okhttp内置的线程池。

读硬盘和读网络需要放在不同的线程池中处理，所以用两个线程池比较合
适。

Glide 必然也需要多个线程池，看下源码是不是这样

```
public final class GlideBuilder { ... private  
GlideExecutor sourceExecutor; //加载源文件的线程池，包括网络加  
载 private GlideExecutor diskCacheExecutor; //加载硬盘缓存  
的线程池 ... private GlideExecutor animationExecutor; //  
动画线程池
```

Glide使用了三个线程池，不考虑动画的话就是两个。

6.2.2 切换线程：

图片异步加载成功，需要在主线程去更新ImageView，

无论是RxJava、EventBus，还是Glide，只要是想从子线程切换到
Android主线程，都离不开Handler。

看下Glide 相关源码：

```
class EngineJob<R> implements  
DecodeJob.Callback<R>, Poolable { ... private static  
final EngineResourceFactory DEFAULT_FACTORY = new  
EngineResourceFactory(); //创建Handler private  
static final Handler MAIN_THREAD_HANDLER = new  
Handler(Looper.getMainLooper(), new  
MainThreadCallback());
```

问RxJava是完全用Java语言写的，那怎么实现从子线程切换到Android
主线程的？依然有很多3-6年的开发答不上来这个很基础的问题，而且
只要是这个问题回答不出来的，接下来有关于原理的问题，基本都答
不上来。

有不少工作了很多年的Android开发不知道鸿洋、郭霖、玉刚说，不知道掘金是个啥玩意，内心估计会想是不是还有叫掘银掘铁的（我不知道有没有）。

我想表达的是，干这一行，真的是需要有对技术的热情，不断学习，不怕别人比你优秀，就怕比你优秀的人比你还努力，而你却不知道。

6.2.3 缓存

我们常说的图片三级缓存：内存缓存、硬盘缓存、网络。

2.3.1 内存缓存

一般都是用 LruCache

Glide 默认内存缓存用的也是LruCache，只不过并没有用Android SDK中的LruCache，不过内部同样是基于LinkHashMap，所以原理是一样的。

```
// -> GlideBuilder#buildif (memoryCache == null) {  
    memoryCache = new  
    LruResourceCache(memorySizeCalculator.getMemoryCacheSize()  
});}
```

既然说到LruCache，必须要了解一下LruCache的特点和源码：

为什么用LruCache？

LruCache 采用最近最少使用算法，设定一个缓存大小，当缓存达到这个大小之后，会将最老的数据移除，避免图片占用内存过大导致OOM。

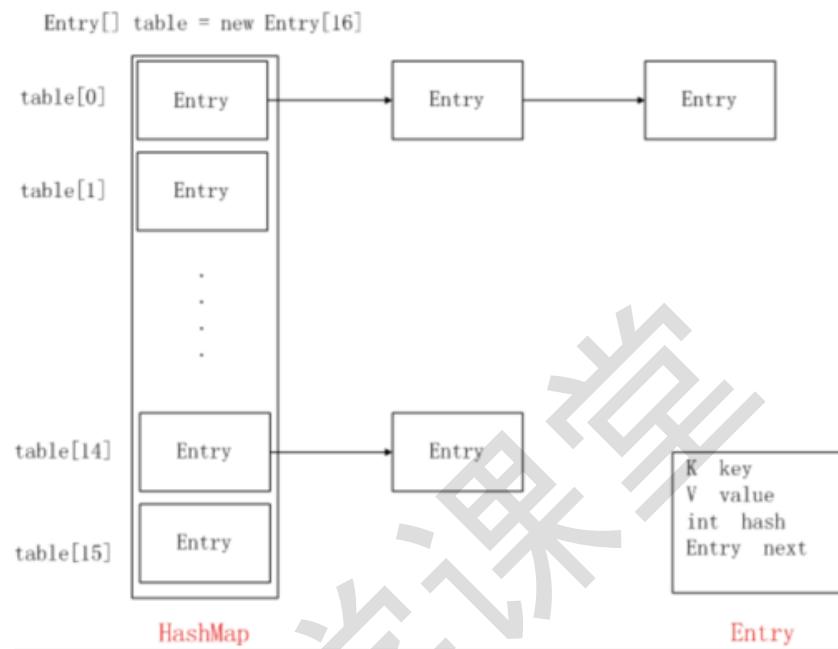
LruCache 源码分析

```
public class LruCache<K, V> {    // 数据最终存在  
    LinkedHashMap 中    private final LinkedHashMap<K, V> map;  
    ...    public LruCache(int maxSize) {        if  
(maxSize <= 0) {            throw new  
IllegalStateException("maxSize <= 0");        }  
        this.maxSize = maxSize;        // 创建一个LinkedHashMap,  
        accessOrder 传true        this.map = new LinkedHashMap<K,  
V>(0, 0.75f, true);    }    ...}
```

LruCache 构造方法里创建一个 LinkedHashMap，accessOrder 参数传 true，表示按照访问顺序排序，数据存储基于 LinkedHashMap。

先看看 LinkedHashMap 的原理吧

LinkedHashMap 继承 HashMap，在 HashMap 的基础上进行扩展，put 方法并没有重写，说明 LinkedHashMap 遵循 HashMap 的数组加链表的结构，



LinkedHashMap 重写了 `createEntry` 方法。

看下 HashMap 的 `createEntry` 方法

```
void createEntry(int hash, K key, V value, int  
bucketIndex) {    HashMapEntry<K,V> e =  
table[bucketIndex];    table[bucketIndex] = new  
HashMapEntry<>(hash, key, value, e);    size++;}
```

HashMap 的数组里面放的是 `HashMapEntry` 对象

看下 LinkedHashMap 的 `createEntry` 方法

```
void createEntry(int hash, K key, V value, int bucketIndex) {    HashMapEntry<K,V> old = table[bucketIndex];    LinkedHashMapEntry<K,V> e = new LinkedHashMapEntry<>(hash, key, value, old); table[bucketIndex] = e; //数组的添加 e.addBefore(header); //处理链表    size++;}
```

LinkedHashMap的数组里面放的是 LinkedHashMapEntry 对象

LinkedHashMapEntry

```
private static class LinkedHashMapEntry<K,V> extends HashMapEntry<K,V> {    // These fields comprise the doubly linked list used for iteration. LinkedHashMapEntry<K,V> before, after; //双向链表 private void remove() {        before.after = after;        after.before = before;    }    private void addBefore(LinkedHashMapEntry<K,V> existingEntry) {        after = existingEntry;        before = existingEntry.before;        before.after = this;        after.before = this;    }
```

LinkedHashMapEntry 继承 HashMapEntry，添加 before 和 after 变量，所以是一个双向链表结构，还添加了 addBefore 和 remove 方法，用于新增和删除链表节点。

LinkedHashMapEntry#addBefore

将一个数据添加到 Header 的前面

```
private void addBefore(LinkedHashMapEntry<K,V> existingEntry) {    after = existingEntry;    before = existingEntry.before;    before.after = this;    after.before = this;}
```

existingEntry 传的都是链表头 header，将一个节点添加到 header 节点前面，只需要移动链表指针即可，添加新数据都是放在链表头 header 的 before 位置，链表头节点 header 的 before 是最新访问的数据，header 的 after 则是最旧的数据。

再看下LinkedHashMapEntry#remove

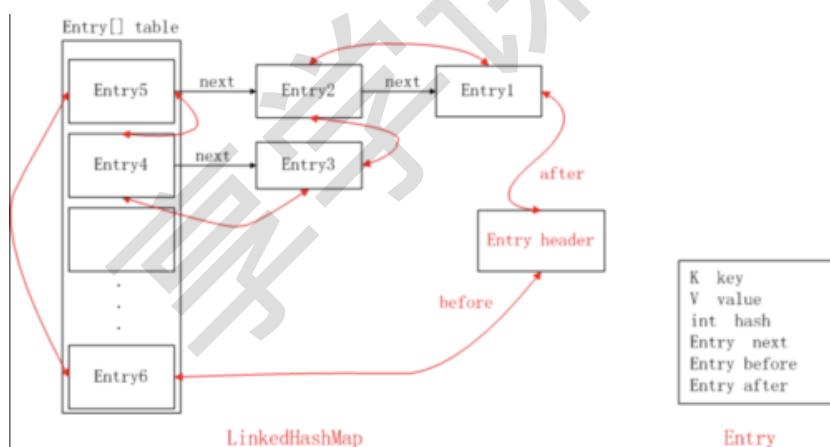
```
private void remove() { before.after = after;
    after.before = before; }
```

链表节点的移除比较简单，改变指针指向即可。

再看下LinkHashMap的put 方法

```
public final V put(K key, V value) { V previous;
synchronized (this) { putCount++; //size增加
    size += safeSizeof(key, value); // 1、
    LinkHashMap的put方法 previous = map.put(key,
value); if (previous != null) { //如果有
    旧的值，会覆盖，所以大小要减掉 size -=
    safeSizeof(key, previous); } }
    trimToSize(maxSize); return previous; }
```

LinkedHashMap 结构可以用这种图表示



LinkHashMap 的 put方法和get方法最后会调用 trimToSize方法，
LruCache 重写 trimToSize方法，判断内存如果超过一定大小，则移除
最老的数据

LruCache#trimToSize，移除最老的数据

```

public void trimToSize(int maxSize) {    while (true) {
    K key;          V value;          synchronized (this) {
        //大小没有超出, 不处理
        if (size <=
maxSize) {           break;
    }
    //超出大小, 移除最老的数据
    = map.eldest();           Map.Entry<K, V> toEvict
    if (toEvict == null) {
        break;
    }
    key =
    toEvict.getKey();           value = toEvict.getValue();
    map.remove(key);           //这个大小的计算,
    safesizeof 默认返回1;           size -= safesizeof(key,
value);           evictionCount++;
    entryRemoved(true, key, value, null);
}
}

```

对LinkHashMap 还不是很理解的话可以参考：

[图解LinkedHashMap原理](#)

LruCache小结：

- LinkHashMap 继承HashMap，在HashMap的基础上，新增了双向链表结构，每次访问数据的时候，会更新被访问的数据的链表指针，具体就是先在链表中删除该节点，然后添加到链表头header之前，这样就保证了链表头header节点之前的数据都是最近访问的（从链表中删除并不是真的删除数据，只是移动链表指针，数据本身在map中的位置是不变的）。
- LruCache 内部用LinkHashMap存取数据，在双向链表保证数据新旧顺序的前提下，设置一个最大内存，往里面put数据的时候，当数据达到最大内存的时候，将最老的数据移除掉，保证内存不超过设定的最大值。

2.3.2 磁盘缓存 DiskLruCache

依赖：

```
implementation 'com.jakewharton:disklrucache:2.0.2'
```

DiskLruCache 跟 LruCache 实现思路是差不多的，一样是设置一个总大小，每次往硬盘写文件，总大小超过阈值，就会将旧的文件删除。简单看下remove操作：

```

    // DiskLruCache 内部也是用LinkedHashMap    private
final LinkedHashMap<String, Entry> lruEntries =
new LinkedHashMap<String, Entry>(0, 0.75f, true);    ...
    public synchronized boolean remove(String key) throws
IOException {        checkNotClosed();
validateKey(key);        Entry entry =
lruEntries.get(key);        if (entry == null ||
entry.currentEditor != null) {            return false;
}        //一个key可能对应多个value, hash冲突的情况
        for (int i = 0; i < valueCount; i++) {            File
file = entry.getCleanFile(i);            //通过
file.delete()        删除缓存文件, 删除失败则抛异常
if
(file.exists() && !file.delete()) {            throw new
IOException("failed to delete " + file);
}
        size -= entry.lengths[i];        entry.lengths[i] =
0;        }        ...        return true; }

```

可以看到 DiskLruCache 同样是利用LinkHashMap的特点，只不过数组里面存的 Entry 有点变化，Editor 用于操作文件。

```

private final class Entry {    private final String key;
    private final long[] lengths;    private boolean
readable;    private Editor currentEditor;    private
long sequenceNumber;    ...}

```

6.2.4 防止OOM

加载图片非常重要的一点是需要防止OOM，上面的LruCache缓存大小设置，可以有效防止OOM，但是当图片需求比较大，可能需要设置一个比较大的缓存，这样的话发生OOM的概率就提高了，那应该探索其它防止OOM的方法。

方法1：软引用

回顾一下Java的四大引用：

- 强引用： 普通变量都属于强引用，比如 `private Context context;`
- 软应用： SoftReference，在发生OOM之前，垃圾回收器会回收 SoftReference引用的对象。

- 弱引用： WeakReference，发生GC的时候，垃圾回收器会回收 WeakReference中的对象。
- 虚引用： 随时会被回收，没有使用场景。

怎么理解强引用：

强引用对象的回收时机依赖垃圾回收算法，我们常说的可达性分析算法，当Activity销毁的时候，Activity会跟GCRoot断开，至于GCRoot是谁？这里可以大胆猜想，Activity对象的创建是在ActivityThread中，ActivityThread要回调Activity的各个生命周期，肯定是持有Activity引用的，那么这个GCRoot可以认为就是ActivityThread，当Activity执行onDestroy的时候，ActivityThread就会断开跟这个Activity的联系，Activity到GCRoot不可达，所以会被垃圾回收器标记为可回收对象。

软引用的设计就是应用于会发生OOM的场景，大内存对象如Bitmap，可以通过 SoftReference 修饰，防止大对象造成OOM，看下这段代码

```
private static LruCache<String,  
SoftReference<Bitmap>> mLruCache = new LruCache<String,  
SoftReference<Bitmap>>(10 * 1024){  
    @Override  
    protected int sizeof(String key, SoftReference<Bitmap>  
value) {  
        //默认返回1，这里应该返回Bitmap占用的内存大  
小，单位：K  
        //Bitmap被回收了，大小是0  
        if  
(value.get() == null){  
            return 0;  
        }  
        return value.get().getByteCount() /1024;  
    }  
};
```

LruCache里存的是软引用对象，那么当内存不足的时候，Bitmap会被回收，也就是说通过SoftReference修饰的Bitmap就不会导致OOM。

当然，这段代码存在一些问题，Bitmap被回收的时候，LruCache剩余的大小应该重新计算，可以写个方法，当Bitmap取出来是空的时候，LruCache清理一下，重新计算剩余内存；

还有另一个问题，就是内存不足时软引用中的Bitmap被回收的时候，这个LruCache就形同虚设，相当于内存缓存失效了，必然出现效率问题。

方法2：onLowMemory

当内存不足的时候，Activity、Fragment会调用onLowMemory方法，可以在这个方法里去清除缓存，Glide使用的就是这种方式来防止OOM。

```
//Glidepublic void onLowMemory() {  
    clearMemory();}public void clearMemory() {    // Engine  
    asserts this anyway when removing resources, fail faster  
    and consistently    util.assertMainThread();    // memory  
    cache needs to be cleared before bitmap pool to clear re-  
    pooled Bitmaps too. See #687.  
    memoryCache.clearMemory();    bitmapPool.clearMemory();  
    arrayPool.clearMemory(); }
```

方法3：从Bitmap 像素存储位置考虑

我们知道，系统为每个进程，也就是每个虚拟机分配的内存是有限的，早期的16M、32M，现在100+M，虚拟机的内存划分主要有5部分：

- 虚拟机栈
- 本地方法栈
- 程序计数器
- 方法区
- 堆

而对象的分配一般都是在堆中，堆是JVM中最大的一块内存，OOM一般都是发生在堆中。

Bitmap之所以占内存大不是因为对象本身大，而是因为Bitmap的像素数据，Bitmap的像素数据大小 = 宽 高 1像素占用的内存。

1像素占用的内存是多少？不同格式的Bitmap对应的像素占用内存是不同的，具体是多少呢？

在Fresco中看到如下定义代码

```
/** * Bytes per pixel definitions */ public static  
final int ALPHA_8_BYTES_PER_PIXEL = 1; public static  
final int ARGB_4444_BYTES_PER_PIXEL = 2; public static  
final int ARGB_8888_BYTES_PER_PIXEL = 4; public static  
final int RGB_565_BYTES_PER_PIXEL = 2; public static  
final int RGBA_F16_BYTES_PER_PIXEL = 8;
```

如果Bitmap使用 `RGB_565` 格式，则1像素占用 2 byte，`ARGB_8888` 格式则占4 byte。

在选择图片加载框架的时候，可以将内存占用这一方面考虑进去，更少的内存占用意味着发生OOM的概率越低。 Glide内存开销是Picasso的一半，就是因为默认Bitmap格式不同。

至于宽高，是指Bitmap的宽高，怎么计算的呢？看 `BitmapFactory.Options` 的 `outWidth`

```
/** * The resulting width of the bitmap. If {@link  
#inJustDecodeBounds} is * set to false, this will be  
width of the output bitmap after any * scaling is  
applied. If true, it will be the width of the input image  
* without any accounting for scaling. * *  
<p>outwidth will be set to -1 if there is an error trying  
to decode.</p> */ public int outwidth;
```

看注释的意思，如果 `BitmapFactory.Options` 中指定 `inJustDecodeBounds` 为true，则为原图宽高，如果是false，则是缩放后的宽高。所以我们一般可以通过压缩来减小Bitmap像素占用内存。

扯远了，上面分析了Bitmap像素数据大小的计算，只是说明Bitmap像素数据为什么那么大。那是否可以让像素数据不放在java堆中，而是放在native堆中呢？据说Android 3.0到8.0 之间Bitmap像素数据存在Java堆，而8.0之后像素数据存到native堆中，是不是真的？看下源码就知道了~

8.0 Bitmap

java层创建Bitmap方法

```
public static Bitmap createBitmap(@Nullable
DisplayMetrics display, int width, int height,
@NonNull Config config, boolean hasAlpha, @NonNull
ColorSpace colorSpace) { ... Bitmap bm;
... if (config != Config.ARGB_8888 || colorSpace == ColorSpace.get(ColorSpace.Named.SRGB)) {
//最终都是通过native方法创建 bm =
nativeCreate(null, 0, width, width, height,
config.nativeInt, true, null, null); } else {
bm = nativeCreate(null, 0, width, width, height,
config.nativeInt, true,
d50.getTransform(), parameters); ...
return bm; }
```

Bitmap 的创建是通过native方法 `nativeCreate`

对应源码

[8.0.0 r4/xref/frameworks/base/core/jni/android/graphics/Bitmap.cpp](https://android.googlesource.com/platform/frameworks/base/+/r4/xref/frameworks/base/core/jni/android/graphics/Bitmap.cpp)

```
//Bitmap.cppstatic const JNINativeMethod gBitmapMethods[]
= { { "nativeCreate",           "
([IIIIIZ[FLandroid/graphics/colorSpace$Rgb$TransferParam
eters;)Landroid/graphics/Bitmap;" ,
(void*)Bitmap_creator },...
```

JNI动态注册，`nativeCreate` 方法 对应 `Bitmap_creator`；

```
//Bitmap.cpp static jobject Bitmap_creator(JNIEnv* env,
jobject, jintArray jColors,
jint offset, jint stride, jint width, jint height,
                jint configHandle, jboolean
isMutable,                      jfloatArray
xyzD50, jobject transferParameters) { ... //1\. 申请
堆内存, 创建native层Bitmap      sk_sp<Bitmap> nativeBitmap =
Bitmap::allocateHeapBitmap(&bitmap, NULL);    if
(!nativeBitmap) {           return NULL; } ... //2.
创建java层Bitmap      return createBitmap(env,
nativeBitmap.release(),
getPremulBitmapCreateFlags(isMutable));}
```

主要两个步骤：

1. 申请内存，创建native层Bitmap，看下allocateHeapBitmap方法

[8.0.0_r4/xref/frameworks/base/libs/hwui/hwui/Bitmap.cpp]
(https://www.androidos.net.cn/android/8.0.0_r4/xref/frameworks/base/libs/hwui/hwui/Bitmap.cpp)

```
//static sk_sp<Bitmap> allocateHeapBitmap(size_t size,
const SkImageInfo& info, size_t rowBytes,
skColorTable* ctable) { // calloc 是c++ 的申请内存函数
void* addr = calloc(size, 1);    if (!addr) {
return nullptr; }    return sk_sp<Bitmap>(new
Bitmap(addr, size, info, rowBytes, ctable));}
```

可以看到通过c++的 `calloc` 函数申请了一块内存空间，然后创建native层Bitmap对象，把内存地址传过去，也就是native层的Bitmap数据（像素数据）是存在native堆中。

1. 创建java 层Bitmap

```
//Bitmap.cppjobject createBitmap(JNIEnv* env, Bitmap*  
bitmap, int bitmapCreateFlags, jbyteArray  
ninePatchChunk, jobject ninePatchInsets, int  
density) { ... BitmapWrapper* bitmapWrapper = new  
BitmapWrapper(bitmap); //通过JNI回调Java层，调用java层的  
Bitmap构造方法 jobject obj = env->  
>NewObject(gBitmap_class, gBitmap_constructorMethodID,  
reinterpret_cast<jlong>(bitmapWrapper), bitmap-  
>width(), bitmap->height(), density,  
isMutable, isPremultiplied, ninePatchChunk,  
ninePatchInsets); ... return obj;}
```

env->NewObject, 通过JNI创建Java层Bitmap对象, gBitmap_class, gBitmap_constructorMethodID这些变量是什么意思, 看下面这个方法, 对应java层的Bitmap的类名和构造方法。

```
//Bitmap.cppint register_android_graphics_Bitmap(JNIEnv*  
env){ gBitmap_class = MakeGlobalRefOrDie(env,  
FindClassOrDie(env, "android/graphics/Bitmap"));  
gBitmap_nativePtr = GetFieldIDOrDie(env, gBitmap_class,  
"mNativePtr", "J"); gBitmap_constructorMethodID =  
GetMethodIDOrDie(env, gBitmap_class, "<init>", "  
(JIIIZZ[BAndroid/graphics/NinePatch$InsetStruct;)V");  
gBitmap_reinitMethodID = GetMethodIDOrDie(env,  
gBitmap_class, "reinit", "(IIZ)V");  
gBitmap_getAllocationByteCountMethodID =  
GetMethodIDOrDie(env, gBitmap_class,  
"getAllocationByteCount", "()I"); return  
android::RegisterMethodsOrDie(env,  
"android/graphics/Bitmap", gBitmapMethods,  
NELEM(gBitmapMethods));}
```

8.0 的Bitmap创建就两个点:

1. 创建native层Bitmap, 在native堆申请内存。
2. 通过JNI创建java层Bitmap对象, 这个对象在java堆中分配内存。

像素数据是存在native层Bitmap，也就是证明8.0的Bitmap像素数据存在native堆中。

7.0 Bitmap

直接看native层的方法，

[/7.0.0 r31/xref/frameworks/base/core/jni/android/graphics/Bitmap.cpp](#)

```
//JNI动态注册static const JNINativeMethod gBitmapMethods[] = { { "nativeCreate", "(I[IIIIIZ)Landroid/graphics/Bitmap;", (void*)Bitmap_creator }, ... static jobject Bitmap_creator(JNIEnv* env, jobject, jIntArray jColors, jint offset, jint stride, jint width, jint height, jint configHandle, jboolean isMutable) { ... //1.通过这个方法来创建native层Bitmap     Bitmap* nativeBitmap = GraphicsJNI::allocateJavaPixelRef(env, &bitmap, NULL);     ...     return GraphicsJNI::createBitmap(env, nativeBitmap, getPremulBitmapCreateFlags(isMutable)); } }
```

native层Bitmap 创建是通过 `GraphicsJNI::allocateJavaPixelRef`，看看里面是怎么分配的， `GraphicsJNI` 的实现类是 [Graphics.cpp](#)

```
android::Bitmap*
GraphicsJNI::allocateJavaPixelRef(JNIEnv* env, SkBitmap* bitmap,
SkColorTable* ctable) {    const SkImageInfo& info =
bitmap->info();    size_t size;    //计算需要的空间大小
if (!computeAllocationSize(*bitmap, &size)) {
return NULL;    }    // we must respect the rowBytes
value already set on the bitmap instead of    //
attempting to compute our own.    const size_t rowBytes =
bitmap->rowBytes();    // 1\. 创建一个数组，通过JNI在java层创
建的    jbyteArray arrayObj = (jbyteArray) env-
>CallObjectMethod(gVMRuntime,
gVMRuntime_newNonMovableArray,
gByte_class, size);
...    // 2\. 获取创建的数组的地址    jbyte* addr = (jbyte*)
env->CallLongMethod(gVMRuntime, gVMRuntime_addressof,
arrayObj);    ...    //3\. 创建Bitmap，传这个地址
android::Bitmap* wrapper = new android::Bitmap(env,
arrayObj, (void*) addr,           info, rowBytes,
ctable);    wrapper->getSkBitmap(bitmap);    // since
we're already allocated, we lockPixels right away    //
HeapAllocator behaves this way too    bitmap-
>lockPixels();    return wrapper;}
```

可以看到，7.0 像素内存的分配是这样的：

1. 通过JNI调用java层创建一个数组
2. 然后创建native层Bitmap，把数组的地址传进去。

由此说明，7.0 的Bitmap像素数据是放在java堆的。

当然，3.0 以下Bitmap像素内存据说也是放在native堆的，但是需要手动释放native层的Bitmap，也就是需要手动调用recycle方法，native层内存才会被回收。这个大家可以自己去看源码验证。

native层Bitmap 回收问题

Java层的Bitmap对象由垃圾回收器自动回收，而native层Bitmap印象中我们是不需要手动回收的，源码中如何处理的呢？

记得有个面试题是这样的：

说说final、finally、finalize 的关系

三者除了长得像，其实没有半毛钱关系，final、finally大家都用的比较多，而 finalize 用的少，或者没用过， finalize 是 Object 类的一个方法，注释是这样的：

```
/**      * Called by the garbage collector on an object  
when garbage collection      * determines that there are  
no more references to the object.      * A subclass  
overrides the {@code finalize} method to dispose of      *  
system resources or to perform other cleanup.      * <p>  
... */ protected void finalize() throws Throwable { }
```

意思是说，垃圾回收器确认这个对象没有其它地方引用到它的时候，会调用这个对象的 finalize 方法，子类可以重写这个方法，做一些释放资源的操作。

在6.0以前，Bitmap 就是通过这个 finalize 方法来释放native层对象的。

[6.0 Bitmap.java](#)

```
Bitmap(long nativeBitmap, byte[] buffer, int width, int
height, int density, boolean isMutable,
boolean requestPremultiplied, byte[]
ninePatchChunk, NinePatch.InsetStruct ninePatchInsets) {
    ... mNativePtr = nativeBitmap; //1.创建 BitmapFinalizer mFinalizer = new
BitmapFinalizer(nativeBitmap); int
nativeAllocationByteCount = (buffer == null ?
getByteCount() : 0);
mFinalizer.setNativeAllocationByteCount(nativeAllocationB
yteCount);} private static class BitmapFinalizer {
private long mNativeBitmap; // Native memory
allocated for the duration of the Bitmap, // if
pixel data allocated into native memory, instead of java
byte[] private int mNativeAllocationByteCount;
BitmapFinalizer(long nativeBitmap) {
mNativeBitmap = nativeBitmap; } public void
setNativeAllocationByteCount(int nativeByteCount) {
if (mNativeAllocationByteCount != 0) {
VMRuntime.getRuntime().registerNativeFree(mNativeAllocati
onByteCount); }
mNativeAllocationByteCount = nativeByteCount;
if (mNativeAllocationByteCount != 0) {
VMRuntime.getRuntime().registerNativeAllocation(mNativeAl
locationByteCount); } }
@Override public void finalize() { try
{ super.finalize(); } catch
(Throwable t) { // Ignore }
finally { //2.就是这里了,
setNativeAllocationByteCount(0);
nativeDestructor(mNativeBitmap);
mNativeBitmap = 0; } } }
```

在Bitmap构造方法创建了一个 `BitmapFinalizer`类，重写`finalize`方法，在java层Bitmap被回收的时候，`BitmapFinalizer`对象也会被回收，`finalize`方法肯定会被调用，在里面释放native层Bitmap对象。

6.0 之后做了一些变化，BitmapFinalizer 没有了，被 [NativeAllocationRegistry](#) 取代。

例如 8.0 Bitmap 构造方法

```
Bitmap(long nativeBitmap, int width, int height, int  
density, boolean isMutable, boolean  
requestPremultiplied, byte[] ninePatchChunk,  
NinePatch.InsetStruct ninePatchInsets) { ...  
mNativePtr = nativeBitmap; long nativeSize =  
NATIVE_ALLOCATION_SIZE + getAllocationByteCount();  
// 创建 NativeAllocationRegistry 这个类，调用  
registerNativeAllocation 方法  
NativeAllocationRegistry registry = new  
NativeAllocationRegistry(  
Bitmap.class.getClassLoader(),  
nativeGetNativeFinalizer(), nativeSize);  
registry.registerNativeAllocation(this, nativeBitmap);  
}
```

NativeAllocationRegistry 就不分析了，不管是 BitmapFinalizer 还是 NativeAllocationRegistry，目的都是在 java 层 Bitmap 被回收的时候，将 native 层 Bitmap 对象也回收掉。一般情况下我们无需手动调用 recycle 方法，由 GC 去盘它即可。

上面分析了 Bitmap 像素存储位置，我们知道，Android 8.0 之后 Bitmap 像素内存在放在 native 堆，Bitmap 导致 OOM 的问题基本不会在 8.0 以上设备出现了（没有内存泄漏的情况下），那 8.0 以下设备怎么办？赶紧升级或换手机吧~

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传
(img-n0KxaBb4-1586487920417)(https://upload-images.jianshu.io/upload_images/18452536-b7c1fbc78c5905b5.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)]

我们换手机当然没问题，但是并不是所有人都能跟上 Android 系统更新的步伐，所以，问题还是要解决~

Fresco之所以能跟Glide正面交锋，必然有其独特之处，文中开头列出Fresco的优点是：“在5.0以下(最低2.3)系统，Fresco将图片放到一个特别的内存区域(Ashmem区)”这个Ashmem区是一块匿名共享内存，Fresco将Bitmap像素放到共享内存去了，共享内存是属于native堆内存。

Fresco关键源码在 `PlatformDecoderFactory` 这个类

```
public class PlatformDecoderFactory { /** * Provide  
the implementation of the PlatformDecoder for the current  
platform using the provided * PoolFactory * *  
@param poolFactory The PoolFactory * @return The  
PlatformDecoder implementation */ public static  
PlatformDecoder buildPlatformDecoder( PoolFactory  
poolFactory, boolean gingerbreadDecoderEnabled) {  
//8.0 以上用 OreoDecoder 这个解码器 if  
(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
int maxNumThreads =  
poolFactory.getFlexByteArrayPoolMaxNumThreads();  
return new OreoDecoder(  
poolFactory.getBitmapPool(), maxNumThreads, new  
Pools.SynchronizedPool<>(maxNumThreads)); } else if  
(Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {  
//大于5.0小于8.0用 ArtDecoder 解码器 int  
maxNumThreads =  
poolFactory.getFlexByteArrayPoolMaxNumThreads();  
return new ArtDecoder(  
poolFactory.getBitmapPool(), maxNumThreads, new  
Pools.SynchronizedPool<>(maxNumThreads)); } else {  
if (gingerbreadDecoderEnabled && Build.VERSION.SDK_INT <  
Build.VERSION_CODES.KITKAT) { //小于4.4 用  
GingerbreadPurgeableDecoder 解码器 return new  
GingerbreadPurgeableDecoder(); } else { //这个  
就是4.4到5.0 用的解码器了 return new  
KitKatPurgeableDecoder(poolFactory.getFlexByteArrayPool()  
); } } }}
```

8.0先不看了，看一下4.4以下是怎么得到Bitmap的，看下
`GingerbreadPurgeableDecoder`这个类有个获取Bitmap的方法

```
//GingerbreadPurgeableDecoderprivate Bitmap
decodeFileDescriptorAsPurgeable(
CloseableReference<PooledByteBuffer> bytesRef,      int
inputLength,      byte[] suffix,
BitmapFactory.Options options) { // MemoryFile : 匿名共
享内存      MemoryFile memoryFile = null;      try {      //将
图片数据拷贝到匿名共享内存      memoryFile =
copyToMemoryFile(bytesRef, inputLength, suffix);
FileDescriptor fd = getMemoryFileDescriptor(memoryFile);
if (mWebpBitmapFactory != null) {      // 创建
Bitmap, Fresco自己写了一套创建Bitmap方法      Bitmap bitmap
= mWebpBitmapFactory.decodeFileDescriptor(fd, null,
options);      return
Preconditions.checkNotNull(bitmap, "BitmapFactory
returned null");      } else {      throw new
IllegalStateException("WebpBitmapFactory is null");
}    }  }
```

捋一捋，4.4以下，Fresco 使用匿名共享内存来保存Bitmap数据，首先将图片数据拷贝到匿名共享内存中，然后使用Fresco自己写的加载Bitmap的方法。

Fresco对不同Android版本使用不同的方式去加载Bitmap，至于4.4-5.0，5.0-8.0，8.0以上，对应另外三个解码器，大家可以从PlatformDecoderFactory这个类入手，自己去分析，思考为什么不同平台要分这么多个解码器，8.0以下都用匿名共享内存不好吗？期待你在评论区跟大家分享~

6.2.5 ImageView 内存泄露

曾经在Vivo驻场开发，带有头像功能的页面被测出内存泄漏，原因是SDK中有个加载网络头像的方法，持有ImageView引用导致的。

当然，修改也比较简单粗暴，将ImageView用WeakReference修饰就完事了。

事实上，这种方式虽然解决了内存泄露问题，但是并不完美，例如在界面退出的时候，我们除了希望ImageView被回收，同时希望加载图片的任务可以取消，队未执行的任务可以移除。

Glide的做法是监听生命周期回调，看 RequestManager 这个类

```
public void onDestroy() {    targetTracker.onDestroy();  
    for (Target<?> target : targetTracker.getAll()) {        //  
        清理任务        clear(target);    }    targetTracker.clear();  
    requestTracker.clearRequests();  
    lifecycle.removeListener(this);  
    lifecycle.removeListener(connectivityMonitor);  
    mainHandler.removeCallbacks(addSelfToLifecycle);  
    glide.unregisterRequestManager(this); }
```

在Activity/fragment 销毁的时候，取消图片加载任务，细节大家可以自己去看源码。

6.2.6 列表加载问题

图片错乱

由于RecyclerView或者ListView的复用机制，网络加载图片开始的时候 ImageView是第一个item的，加载成功之后ImageView由于复用可能跑到第10个item去了，在第10个item显示第一个item的图片肯定是错的。

常规的做法是给ImageView设置tag，tag一般是图片地址，更新 ImageView之前判断tag是否跟url一致。

当然，可以在item从列表消失的时候，取消对应的图片加载任务。要考虑放在图片加载框架做还是放在UI做比较合适。

线程池任务过多

列表滑动，会有很多图片请求，如果是第一次进入，没有缓存，那么队列会有很多任务在等待。所以在请求网络图片之前，需要判断队列中是否存在该任务，存在则不加到队列去。

总结

本文通过Glide开题，分析一个图片加载框架必要的需求，以及各个需求涉及到哪些技术和原理。

- 异步加载：最少两个线程池
- 切换到主线程：Handler
- 缓存：LruCache、DiskLruCache，涉及到LinkHashMap原理
- 防止OOM：软引用、LruCache、图片压缩没展开讲、Bitmap像素存储位置源码分析、Fresco部分源码分析
- 内存泄露：注意ImageView的正确引用，生命周期管理
- 列表滑动加载的问题：加载错乱用tag、队满任务存在则不添加

第七节 Glide OOM问题解决方法汇总

- 1、引入largeHeap属性，让系统为App分配更多的独立内存。
- 2、禁止Glide内存缓存。设置skipMemoryCache(true)。
- 3、自定义GlideModule。设置MemoryCache和BitmapPool大小。
- 4、升级到Glide4.0，使用asDrawable代替asBitmap，drawable更省内存。
- 5、ImageView的scaleType为fitXY时，改为fitCenter/centerCrop/fitStart/fitEnd显示。
- 6、不使用application作为context。当context为application时，会把imageView的生命周期延长到整个运行过程中，imageView不能被回收，从而造成OOM异常。
- 7、使用application作为context。但是对ImageView使用弱引用或软引用，尽量使用SoftReference，当内存不足时，将及时回收无用的ImageView。
- 8、当列表在滑动的时候，调用Glide的pauseRequests()取消请求，滑动停止时，调用resumeRequests()恢复请求。
- 9、Try catch某些大内存分配的操作。考虑在catch里面尝试一次降级的内存分配操作。例如decode bitmap的时候，catch到OOM，可以尝试把采样比例再增加一倍之后，再次尝试decode。
- 10、BitmapFactory.Options和BitmapFactory.decodeStream获取原始图片的宽、高，绕过Java层加载Bitmap，再调用Glide的override(width,height)控制显示。

11、图片局部加载。参考：SubsamplingScaleImageView，先将图片下载到本地，然后去加载，只加载当前可视区域，在手指拖动的时候再去加载另外的区域。

第八节OkHttp源码分析

【面试问题】

如何使用OkHttp进行异步网络请求，并根据请求结果刷新UI？

可否介绍一下OkHttp的整个异步请求流程？

OkHttp对于网络请求都有哪些优化，如何实现的？

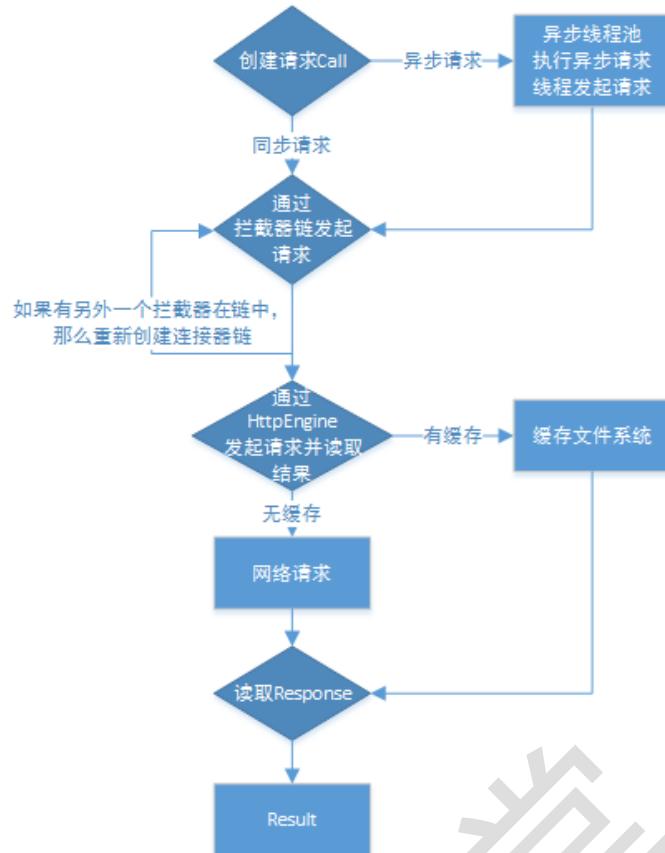
OkHttp框架中都用到了哪些设计模式？

8.1 OkHttp源码解析

本来这篇文章上个星期就写好了，但是当时准备上传的时候，公司停电了，而且没保存，所以，没办法，又得重新写了。说一下我个人的想法啊，很多人觉得看源码特别难，无从下手，很多源码看不懂。我觉得这是很正常的事，除非某个框架代码是你写的，

不然，你很难把每一句代码都搞懂，就连框架的作者，时间一久，都没有办法保证说每一句代码都能够看得懂。我们花个三四天，一两个星期，都很难把我们自己公司新接手的项目熟悉完，更何况是这些框架了。我说一下我平时是怎么看框架源码的吧，每一个框架，首先，会用，这是最基本的，连用都不会用，还别谈其他的了。其次，每一个框架，基本上都会有流程图，这些流程图，在网上都有。我们找到这个流程图，这个流程图，基本上都是整个框架的主干，我们顺着这个主干阅读，你就会发现，很明了，很清晰。当你把主体熟悉完以后，如果，你想更加深入的了解，你就可以在这个主体基础上慢慢的延伸出去。好了，废话不多说了，今天我们来了解一下OkHttp，这里是基于OkHttp3的源码，OkHttp2.x的源码和OkHttp3的会有点区别。我们按照上面的说的方式来阅读一下。

OkHttp流程图



OkHttp基本使用

gradle依赖

```
implementation 'com.squareup.okhttp3:okhttp:3.11.0'  
implementation 'com.squareup.okio:okio:1.15.0'
```

```
/*这里拿get请求来      异步的get请求
 */
public void okhttpAsyn() {          //设置超时的时间
    OkHttpClient.Builder builder = new
    OkHttpClient.Builder()
    .connectTimeout(15, TimeUnit.SECONDS)
    .writeTimeout(20, TimeUnit.SECONDS)
    .readTimeout(20, TimeUnit.SECONDS);           OkHttpClient
    OkHttpClient = builder.build();           Request request =
    new Request.Builder()                      .get() //设置请求模式
                                                .url("https://www.baidu.com/") .build();
    Call call = okHttpClient.newCall(request);   call.enqueue(new
    CallBack() { @Override                     public void
    onFailure(Call call, IOException e) {
        Log.d("MainActivity", "-----onFailure-----");
    } @Override                     public void onResponse(Call
    call, Response response) throws IOException {
        Log.d("MainActivity", "----onResponse----" +
        response.body().toString());
        runOnUiThread(new Runnable() { @Override
            public void run() {
                Toast.makeText(MainActivity.this, "请求成功",
                Toast.LENGTH_LONG).show(); } } ); } });
}

```

OkHttp源码分析

从OkHttp的基本使用中，我们看到，通过okHttpClient.newCall()方法，拿到这个call对象，我们看看newCall是怎么走的

```
/* Prepares the {@code request} to be
executed at some point in the future. 3 */
@Override public Call newCall(Request request) {
    return RealCall.newRealCall(this, request, false /* for
    web socket */); } static RealCall
newRealCall(OkHttpClient client, Request originalRequest,
boolean forWebSocket) { // Safely publish the call
instance to the EventListener. RealCall call = new
RealCall(client, originalRequest, forWebSocket);
call.eventListener =
client.eventListenerFactory().create(call); return
call; }
```

从这里的源码知道，`okHttpClient.newCall()`实际上返回的是`RealCall`对象，而`call.enqueue()`，实际上是调用的了`RealCall`中的`enqueue()`方法，我们看看`enqueue()`方法方法怎么走。

```
@Override public void enqueue(CallBack responseCallback)
{ synchronized (this) { if (executed) throw new
IllegalStateException("Already Executed"); executed
= true; } captureCallStackTrace();
eventListener.callStart(this);
client.dispatcher().enqueue(new
AsyncCall(responseCallback)); }
```

可以看到`client.dispatcher().enqueue(new AsyncCall(responseCallback))`;这句代码，也就是说，最终是有的请求是有`dispatcher`来完成，我们看看`dispatcher`。

想
識
思
學

```
/* Copyright (C) 2013 Square, Inc. */
Licensed under the Apache License, Version 2.0 (the
"License"); you may not use this file except in
compliance with the License. You may obtain a copy
of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in
writing, software distributed under the License is
distributed on an "AS IS" BASIS, WITHOUT WARRANTIES
OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing
permissions and limitations under the License.

package okhttp3; import java.util.ArrayDeque;
import java.util.ArrayList; import
java.util.Collections; import java.util.Deque; import
java.util.Iterator; import java.util.List; import
java.util.concurrent.ExecutorService; import
java.util.concurrent.SynchronousQueue; import
java.util.concurrent.ThreadPoolExecutor; import
java.util.concurrent.TimeUnit; import
javax.annotation.Nullable; import
okhttp3.RealCall.AsyncCall; import
okhttp3.internal.Util; /** * Policy on when async
requests are executed. * <p>Each dispatcher uses an
{@link ExecutorService} to run calls internally. If you
supply your own executor, it should be able to run
{@linkplain #getMaxRequests the configured maximum}
number of calls concurrently. */
public final
class Dispatcher { //最大请求的并发数 private int
maxRequests = 64; //每个主机最大请求数 private int
maxRequestsPerHost = 5; private @Nullable Runnable
idlecallback; /** 消费线程池 */ private @Nullable
ExecutorService executorService; 48 /** 准备运行的异步请
求队列 */
private final Deque<AsyncCall>
readyAsyncCalls = new ArrayDeque<>(); /** 正在运行的异步请
求队列 */
private final Deque<AsyncCall>
runningAsyncCalls = new ArrayDeque<>(); /** 正在运行的同
步请求队列 */
private final Deque<RealCall>
```

```
runningSyncCalls = new ArrayDeque<>();    /** 构造方法 */
public Dispatcher(ExecutorService executorService) {
    this.executorService = executorService;    }    public
Dispatcher() {    }    public synchronized
ExecutorService executorService() {    if
(executorService == null) {        executorService = new
ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60,
TimeUnit.SECONDS,        new
SynchronousQueue<Runnable>(), Util.threadFactory("okHttp
Dispatcher", false));    }    return executorService;
}
/** *      *设置并发执行最大的请
求数量      * <p>If more than {@code maxRequests} requests
are in flight when this is invoked, those requests
* will remain in flight. */    public synchronized void
setMaxRequests(int maxRequests) {    if (maxRequests <
1) {        throw new IllegalArgumentException("max < 1:
" + maxRequests);    }    this.maxRequests =
maxRequests;    promoteCalls();    }    //获取到最大请
求的数量    public synchronized int getMaxRequests() {
return maxRequests;    }    /** * 设置每个主机并发执行
的请求的最大数量      * <p>If more than {@code
maxRequestsPerHost} requests are in flight when this is
invoked, those      * requests will remain in flight.
*      * <p>WebSocket connections to hosts <b>do not</b>
count against this limit.*/    public synchronized void
setMaxRequestsPerHost(int maxRequestsPerHost) {    if
(maxRequestsPerHost < 1) {        throw new
IllegalArgumentException("max < 1: " +
maxRequestsPerHost);    }    this.maxRequestsPerHost =
maxRequestsPerHost;    promoteCalls();    }    //获取每个主机最大
并发数量    public synchronized int getMaxRequestsPerHost()
{    return maxRequestsPerHost;    }    /*** Set a callback
to be invoked each time the dispatcher becomes idle (when
the number of running* calls returns to zero).** <p>Note:
The time at which a {@linkplain Call call} is considered
idle is different depending* on whether it was run
{@linkplain Call#enqueue(Call) asynchronously} or*
{@linkplain Call#execute() synchronously}. Asynchronous
```

```
calls become idle after the* {@link Callback#onResponse
onResponse} or {@link Callback# onFailure onFailure}
callback has* returned. Synchronous calls become idle
once {@link Call#execute() execute()} returns. This*
means that if you are doing synchronous calls the network
layer will not truly be idle until* every returned {@link
Response} has been closed.*/ public synchronized void
setIdleCallback(@Nullable Runnable idleCallback) {
    this.idleCallback = idleCallback; } synchronized void
enqueue(AsyncCall call) { if
(runningAsyncCalls.size() < maxRequests &&
runningCallsForHost(call) < maxRequestsPerHost) {
    runningAsyncCalls.add(call);
    executorService().execute(call); } else {
    readyAsyncCalls.add(call); } } /*** Cancel all calls
currently enqueued or executing. Includes calls executed
both {@link PlainCall#execute() synchronously} and
{@link PlainCall#enqueue asynchronously}.*/ public
synchronized void cancelAll() { for (AsyncCall call :
readyAsyncCalls) { call.get().cancel(); } for
(AsyncCall call : runningAsyncCalls) {
    call.get().cancel(); } for (RealCall call :
runningSyncCalls) { call.cancel(); } } private void
promoteCalls() { if (runningAsyncCalls.size() >=
maxRequests) return; // Already running max capacity.
if (readyAsyncCalls.isEmpty()) return; // No ready calls
to promote. for (Iterator<AsyncCall> i =
readyAsyncCalls.iterator(); i.hasNext(); ) {
    AsyncCall call = i.next(); if
(runningCallsForHost(call) < maxRequestsPerHost) {
        i.remove(); runningAsyncCalls.add(call);
        executorService().execute(call); } if
(runningAsyncCalls.size() >= maxRequests) return; //
Reached max capacity. } } //-----省略若干代码-----
----- }
```

我们来找到这段代码

```
synchronized void enqueue(AsyncCall call) {  
    if (runningAsyncCalls.size() < maxRequests &&  
runningCallsForHost(call) < maxRequestsPerHost) {  
runningAsyncCalls.add(call);  
executorService().execute(call);  
} else {  
readyAsyncCalls.add(call);  
}  
}
```

当正在运行的异步请求队列中的数量小于64并且正在运行的请求主机数小于5时则把请求加载到runningAsyncCalls中并在线程池中执行，否则就再进入到readyAsyncCalls中进行缓存等待。而runningAsyncCalls这个请求队列存放的就是AsyncCall对象，而这个AsyncCall就是RealCall的内部类，也就是说executorService().execute(call);实际上走的是RealCall类中的execute()方法。

```
1 @Override protected void execute() { 2     boolean  
signalledCallback = false;  
3     try { 4         Response response =  
getResponsewithInterceptorChain(); 5             if  
(retryAndFollowUpInterceptor.isCanceled()) { 6  
signalledCallback = true;  
7             responseCallback.onFailure(RealCall.this,  
new IOException("Canceled"));  
8         } else { 9             signalledCallback = true;  
10             responseCallback.onResponse(RealCall.this,  
response);  
11     }  
12     } catch (IOException e) {  
13         if (signalledCallback) {  
14             // Do not signal the callback twice!  
15             Platform.get().log(INFO, "callback failure  
for " + toLoggableString(), e);  
16         } else {  
17             eventListener.callFailed(RealCall.this, e);  
}}
```

```
18         responseCallback onFailure(RealCall.this,
19 );
20     } finally {
21         client.dispatcher().finished(this);
22 }
23 }
```

这部分的代码，相信很多人都能够看的明白，无非就是一些成功，失败的回调，这段代码，最重要的是esponse response =
getResponseWithInterceptorChain();和
client.dispatcher().finished(this);我们先来看看
client.dispatcher().finished(this);这句代码是怎么执行的。

```

1 /* Used by {@code AsyncCall#run} to signal
completion. */
2     void finished(AsyncCall call) {
3         finished(runningAsyncCalls, call, true);
4     }
5
/* Used by {@code Call#execute} to signal completion.
*/
6     void finished(RealCall call) {
7         finished(runningSyncCalls, call, false);
8     }
9
10    private <T> void finished(Deque<T> calls, T call, boolean
11        promoteCalls) {
12        int runningCallsCount;
13        Runnable
14        idleCallback;
15        synchronized (this) {
16            if (!calls.remove(call))
17                throw new AssertionError("Call
18                    wasn't in-flight!");
19            if (promoteCalls)
20                runningCallsCount =
21                    runningCallsCount();
22            idleCallback =
23                this.idleCallback;
24        }
25    }
26    private void promoteCalls() {
27        if
28            (runningAsyncCalls.size() >= maxRequests)
29                return; // Already running max capacity.
30        if
31            (readyAsyncCalls.isEmpty())
32                return; // No ready calls to
33        promote.
34        for (Iterator<AsyncCall> i =
35            readyAsyncCalls.iterator(); i.hasNext(); ) {
36            AsyncCall call = i.next();
37            if
38                (runningCallsForHost(call) < maxRequestsPerHost)
39                i.remove();
40            runningAsyncCalls.add(call);
41            executorService().execute(call);
42        }
43        if
44            (runningAsyncCalls.size() >= maxRequests)
45                return; // Reached max capacity.
46    }

```

由于client.dispatcher().finished(this);这句代码是放到finally中执行的，所以无论什么情况，都会执行上面的promoteCalls()方法，而从promoteCalls()方法中可以看出通过遍历来获取到下一个请求从而执行下一个网络请求。

回过头来，我们看看这一句代码Response response =
getResponseBodyWithInterceptorChain(); 通过
getResponseBodyWithInterceptorChain();来获取到response，然后回调返
回。很明显getResponseBodyWithInterceptorChain()这句代码里面进行了网

络请求。我们看看是怎么执行的。

```
1 Response getResponseWithInterceptorChain() throws
IOException { 2      // Build a full stack of
interceptors. 3      List<Interceptor> interceptors = new
ArrayList<>(); 4
interceptors.addAll(client.interceptors()); 5
interceptors.add(retryAndFollowUpInterceptor); 6
interceptors.add(new
BridgeInterceptor(client.cookieJar())); 7
interceptors.add(new
CacheInterceptor(client.internalCache())); 8
interceptors.add(new ConnectInterceptor(client)); 9
if (!forWebSocket) {10
interceptors.addAll(client.networkInterceptors());11 }12
    interceptors.add(new
CallServerInterceptor(forWebSocket));13 14
Interceptor.Chain chain = new
RealInterceptorChain(interceptors, null, null, null, 0,15
    originalRequest, this, eventListener,
client.connectTimeoutMillis(),16
client.readTimeoutMillis(),
client.writeTimeoutMillis());17 18      return
chain.proceed(originalRequest);19 }20 }
```

从上面代码可以知道，缓存，网络请求，都封装成拦截器的形式。拦截器主要用来观察，修改以及可能短路的请求输出和响应的回来。最后return chain.proceed，而chain是通过new RealInterceptorChain来获取到的，我们来看看RealInterceptorChain对象，然后找到proceed()方法。

```
1 public Response proceed(Request request,
StreamAllocation streamAllocation, HttpCodec httpCodec, 2
    RealConnection connection) throws IOException { 3
    if (index >= interceptors.size()) throw new
AssertionError(); 4 5      calls++; 6 7      // If we
already have a stream, confirm that the incoming request
will use it. 8      if (this.httpCodec != null &&
!this.connection.supportsUrl(request.url())) { 9
throw new IllegalStateException("network interceptor " +
interceptors.get(index - 1)10                  + " must retain
the same host and port");11 }12 13      // If we already
have a stream, confirm that this is the only call to
chain.proceed().14      if (this.httpCodec != null &&
calls > 1) {15      throw new
IllegalStateException("network interceptor " +
interceptors.get(index - 1)16                  + " must call
proceed() exactly once");17 }18 19      // 调用下一个拦截器20
    RealInterceptorChain next = new
RealInterceptorChain(interceptors, streamAllocation,
httpCodec,21                  connection, index + 1, request,
call, eventListener, connectTimeout, readTimeout,22
writeTimeout);23      Interceptor interceptor =
interceptors.get(index);24      Response response =
interceptor.intercept(next); //调用拦截器中的intercept()方法
25 26      // Confirm that the next interceptor made its
required call to chain.proceed().27      if (httpCodec !=
null && index + 1 < interceptors.size() && next.calls !=
1) {28      throw new IllegalStateException("network
interceptor " + interceptor29                  + " must call
proceed() exactly once");30 }31 32      // Confirm that
the intercepted response isn't null.33      if (response
== null) {34      throw new
NullPointerException("interceptor " + interceptor + "
returned null");35 }36 37      if (response.body() ==
null) {38      throw new IllegalStateException(39
    "interceptor " + interceptor + " returned a response
with no body");40 }41 42      return response;43  }
```

从上面的代码可以看出来，chain.proceed主要是讲集合中的拦截器遍历出来，然后通过调用每一个拦截器中的intercept()方法，然后获取到response结果，返回。

我们看看CacheInterceptor这个类，找到intercept()方法。

```
1  @Override public Response intercept(Chain chain)
throws IOException { 2      Response cacheCandidate =
cache != null ? cache.get(chain.request()) 4
: null; 5 6      long now =
System.currentTimeMillis(); 7 8      //创建
CacheStrategy.Factory对象，进行缓存配置 9      CacheStrategy
strategy = new CacheStrategy.Factory(now,
chain.request(), cacheCandidate).get();10     //网络请求11
Request networkRequest = strategy.networkRequest;12
//缓存响应13      Response cacheResponse =
strategy.cacheResponse;14 15      if (cache != null) {16
//记录当前请求是网络发起还是缓存发起17
cache.trackResponse(strategy);18 }19 20      if
(cacheCandidate != null && cacheResponse == null) {21
closeQuietly(cacheCandidate.body()); // The cache
candidate wasn't applicable. Close it.22 }23 24      // 不
进行网络请求并且缓存不存在或者过期则返回504错误25      if
(networkRequest == null && cacheResponse == null) {26
return new Response.Builder()27
.request(chain.request())28
.protocol(Protocol.HTTP\_1\_1)29           .code(504)30
.message("Unsatisfiable Request (only-if-
cached)")31 .body(Util.EMPTY\_RESPONSE)32
.sentRequestAtMillis(-1L)33
.receivedResponseAtMillis(System.currentTimeMillis())34
.build();35 }36 37      // 不进行网络请求，而且缓存可以使用，直接
返回缓存38      if (networkRequest == null) {39      return
cacheResponse.newBuilder()40
.cacheResponse(stripBody(cacheResponse))41 .build();42
}43 44      //进行网络请求    45      Response
networkResponse = null;46      try {47
networkResponse = chain.proceed(networkRequest);48      }
finally {49      // If we're crashing on I/O or
otherwise, don't leak the cache body.50      if
(networkResponse == null && cacheCandidate != null) {51
closeQuietly(cacheCandidate.body());52 }53 }54 55
//\-----省略若干代码-----56 57      return
response;58 }
```

上面我做了很多注释，基本的流程是有缓存就取缓存里面的，没有缓存就请求网络。我们来看看网络请求的类CallServerInterceptor

基础课
基础课

想
識
思
學

```
 1 /* 2 * Copyright (C) 2016 Square, Inc. 3 */ 4
/* Licensed under the Apache License, Version 2.0 (the
"License"); 5 * you may not use this file except in
compliance with the License. 6 * You may obtain a copy
of the License at 7 */ 8 */
http://www.apache.org/licenses/LICENSE-2.0 9 */ 10 */
Unless required by applicable law or agreed to in
writing, software 11 */ distributed under the License is
distributed on an "AS IS" BASIS, 12 */ WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied. 13 */ See the License for the specific language
governing permissions and 14 */ limitations under the
License. 15 */ 16 package okhttp3.internal.http; 17 18
import java.io.IOException; 19 import
java.net.ProtocolException; 20 import
okhttp3.Interceptor; 21 import okhttp3.Request; 22 import
okhttp3.Response; 23 import okhttp3.internal.util; 24
import okhttp3.internal.connection.RealConnection; 25
import okhttp3.internal.connection.StreamAllocation; 26
import okio.Buffer; 27 import okio.BufferedSink; 28
import okio.ForwardingSink; 29 import okio.Okio; 30
import okio.Sink; 31 32 /* This is the last
interceptor in the chain. It makes a network call to the
server. */ 33 public final class CallServerInterceptor
implements Interceptor { 34     private final boolean
forWebSocket; 35 36     public
CallServerInterceptor(boolean forWebSocket) { 37
this.forWebSocket = forWebSocket; 38 } 39 40
@Override public Response intercept(Chain chain) throws
IOException { 41         RealInterceptorChain realChain =
(RealInterceptorChain) chain; 42         HttpCodec httpCodec
= realChain.httpStream(); 43         StreamAllocation
streamAllocation = realChain.streamAllocation(); 44
RealConnection connection = (RealConnection)
realChain.connection(); 45         Request request =
realChain.request(); 46 47         long sentRequestMillis =
System.currentTimeMillis(); 48 49
realChain.eventListener().requestHeadersStart(realChain.c
```

```
all()); 50     httpCodec.writeRequestHeaders(request); 51

realChain.eventListener().requestHeadersEnd(realChain.cal
1(), request); 52 53     Response.Builder
responseBuilder = null; 54     if
 HttpMethod.permitsRequestBody(request.method()) &&
request.body() != null) { 55         // If there's a
"Expect: 100-continue" header on the request, wait for a
"HTTP/1.1 100 56         // Continue" response before
transmitting the request body. If we don't get that,
return 57         // what we did get (such as a 4xx
response) without ever transmitting the request body. 58
    if ("100-
continue".equalsIgnoreCase(request.header("Expect"))) {
59         httpCodec.flushRequest(); 60
realChain.eventListener().responseHeadersStart(realChain.
call()); 61     responseBuilder =
httpCodec.readResponseHeaders(true); 62             } 63 64
    if (responseBuilder == null) { 65         // write
the request body if the "Expect: 100-continue"
expectation was met. 66
realChain.eventListener().requestBodyStart(realChain.call
()); 67     Long contentLength =
request.body().contentLength(); 68             CountingSink
requestBodyOut = 69             new
CountingSink(httpCodec.createRequestBody(request,
contentLength)); 70             BufferedSink
bufferedRequestBody = Okio.buffer(requestBodyOut); 71 72
    request.body().writeTo(bufferedRequestBody); 73
    bufferedRequestBody.close(); 74
realChain.eventListener() 75
.requestBodyEnd(realChain.call(),
requestBodyOut.successfulCount); 76             } else if
(!connection.isMultiplexed()) { 77         // If the
"Expect: 100-continue" expectation wasn't met, prevent
the HTTP/1 connection 78         // from being reused.
Otherwise we're still obligated to transmit the request
body to 79             // leave the connection in a
```

```
consistent state. 80
streamAllocation.noNewStreams(); 81      } 82      } 83
84      httpCodec.finishRequest(); 85  86      if
(responseBuilder == null) { 87
realChain.eventListener().responseHeadersStart(realChain.
call()); 88      responseBuilder =
httpCodec.readResponseHeaders(false); 89      } 90  91
Response response = responseBuilder 92
.request(request) 93
.handshake(streamAllocation.connection().handshake()) 94
    .sentRequestAtMillis(sentRequestMillis) 95
    .receivedResponseAtMillis(System.currentTimeMillis())
96      .build(); 97  98      int code =
response.code(); 99      if (code == 100) {100      //
server sent a 100-continue even though we did not request
one.101      // try again to read the actual response102
    responseBuilder =
httpCodec.readResponseHeaders(false);103 104
response = responseBuilder105 .request(request)106
.handshake(streamAllocation.connection().handshake())107
.sentRequestAtMillis(sentRequestMillis)108
.receivedResponseAtMillis(System.currentTimeMillis())109
.build();110 111      code = response.code();112 }113
114 realChain.eventListener()115
.responseHeadersEnd(realChain.call(), response);116 117
    if (forWebSocket && code == 101) {118      //
Connection is upgrading, but we need to ensure
interceptors see a non-null response body.119
response = response.newBuilder()120
.body(Util.EMPTY_RESPONSE)121 .build();122      } else
{123      response = response.newBuilder()124
.body(httpCodec.openResponseBody(response))125
.build();126 }127 128      if
("close".equalsIgnoreCase(response.request().header("Conn
ection")))129      ||
"close".equalsIgnoreCase(response.header("Connection")))
{130 streamAllocation.noNewStreams();131 }132 133      if
((code == 204 || code == 205) &&
```

```
response.body().contentLength() > 0) {134         throw new  
ProtocolException(135                 "HTTP " + code + " had  
non-zero Content-Length: " +  
response.body().contentLength());136 }137 138     return  
response;139 }140 }
```

到这里，基本上okhttp的整个流程就出来了，当然，这里只是一个整体的大概流程，如果要抠的很细，那就不是一篇文章能够说明的了了.....现在回过头来再看一眼流程图，是不是感觉特别明朗了。

8.2 Okhttp面试简答

简略回答：

1.1、简单说一下okhttp。

- 1、支持SPDY、HTTP2.0
- 2、无缝支持GZIP来减少数据流量
- 3、支持同步、异步（异步使用较多）
- 4、缓存响应数据来减少重复的网络请求
- 5、可以从很多常用的连接问题中自动恢复

1.2、Okhttp的核心类有哪些？简单讲一下

Dispatcher类：

Interceptor类：

1.3、OkHttp方面的其他面试题

- 1、如何使用OkHttp进行异步网络请求，并根据请求结果刷新UI
- 2、可否介绍一下OkHttp的整个异步请求流程
- 3、OkHttp对于网络请求都有哪些优化，如何实现的
- 4、OkHttp框架中都用到了哪些设计模式

详细回答：

2.1、简单说一下okhttp。

- 1、支持SPDY、HTTP2.0

在早期的互联网中，由于协议都是一些比较简单的协议，内容基本上都是一些静态的页面、图片等，所以无连接、无状态的HTTP可以发挥自己简单快速、灵活的优势。但随着业务逻辑越来越复杂以及我们对安全性的重视，无连接、无状态反而成为了HTTP的劣势，所以也就又来后来更加高级的互联网协议的诞生。

第一个要讲的就是SPDY，这是谷歌开发的一种互联网协议。它是一种HTTP的兼容协议、支持多路复用请求、对请求划分优先级（优先返回文字，图片音频等随后返回）、压缩HTTP头，以减少请求数量。

而HTTP2.0是在SPDY的基础上开发而来的，那么既然有了SPDY，为什么还要开发HTTP2.0。这是因为SPDY是完全由谷歌公司开发，这么重要的网络协议被把持在一家公司手里显然是不合适的。所以IETF（国际互联网工程任务组 The Internet Engineering Task Force，简称 IETF）就重新开发了HTTP2.0。HTTP2.0在SPDY的基础上又添加了更安全的SSL协议。SPDY和HTTP2.0作为一种更高级的网络协议，Okhttp必须得支持才行啊。而像其他的框架volley暂时是不支持HTTP/2的。所以这可以作为它的一个优点！

2、无缝支持GZIP来减少数据流量

为什么叫无缝支持的，意思就是说，你发送的数据和接受的收据在传递过程中都是经过gzip压缩的，并且这基本上你不需要你手动处理的，框架自动会帮你处理好。

首先，我们给request的请求头添加了("Accept-Encoding", "gzip")的键值对，说明发送的请求数据是经过gzip压缩的；

其次，我们在处理从服务器返回的数据时也会判断：

```
"gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))
```

这两个步骤都是在BridgeIntercept的intercept方法中设置的。

3、支持同步、异步（异步使用较多）

4、缓存响应数据来减少重复的网络请求
CacheInterceptor.java 这个简单，如题

5、可以从很多常用的连接问题中自动恢复

RetryAndFollowUpInterceptor：重试重定向拦截器

- a、请求失败后重新尝试连接：从Retry这个单词理解，但是在OKHttp中并不是所有的请求失败后（即返回码不是200）都会去重新连接，而是在发生RouteException或者IOException后再根据一些策略进行一些判断，如果可以恢复，就重新进请求
- b、继续请求：FollowUp本意是跟进的意思，主要有以下几种类型可以继续发起请求：部分以3或4开头的返回码的情况下可以继续发送请求。

注意：其中FollowUp的次数受到限制，OKHTTP内部限制次数为20次以内

2.2、Okhttp的核心类有哪些？简单讲一下

Dispatcher类：

Dispatcher通过维护一个线程池，来维护、管理、执行OKHttp的请求。Dispatcher内部维护着三个队列：同步请求队列runningSyncCalls、异步请求队列runningAsyncCalls、异步缓存队列readyAsyncCalls，和一个线程池executorService。

Dispatcher类整体可以参照生产者消费者模式来理解：

Dispatcher是生产者，executorService是消费者池，runningSyncCalls、runningAsyncCalls和readyAsyncCalls是消费者，用来消费请求Call。

Interceptor类：

官网：拦截器是Okhttp中提供的一种强大机制，它可以实现网络监听、请求、以及响应重写、请求失败重试等功能。

RetryAndFollowUpInterceptor、BridgeIntercept、CacheIntercept、ConnectIntercept、CallServerIntercept

- RetryAndFollowUpInterceptor：重试和失败重定向拦截器
- BridgeInterceptor：桥接拦截器，处理一些必须的请求头信息的拦截器
- CacheInterceptor：缓存拦截器，用于处理缓存

- ConnectInterceptor: 连接拦截器，建立可用的连接，是 CallServerInterceptor的基本
- CallServerInterceptor: 请求服务器拦截器将 http 请求写进 IO 流当中，并且从 IO 流中读取响应 Response

2.3、OkHttp方面的其他面试题

2.3.1、如何使用OkHttp进行异步网络请求，并根据请求结果刷新UI

第一步，创建一个OkHttpClient对象 OkHttpClient mClient = new OkHttpClient.Builder().build();

第二步，创建携带请求信息的Request对象 Request request = new Request.Builder().url("http://www.baidu.com").get().build();

第三步，创建Call对象 Call call = mClient.newCall(request);

第四步，call.enqueue()

需要注意的是，不能直接在Callback中更新UI，否则会报出异常

2.3.2、OkHttp对于网络请求都有哪些优化，如何实现的

a、通过连接池来减少请求延时：

我们知道，在okhttp中，我们每次的request请求都可以理解为一个connection，而每次发送请求的时候我们都要经过tcp的三次握手，然后传输数据，最后在释放连接。在高并发或者多个客户端请求的情况下，多次创建就会导致性能低下。如果能够connection复用的话，就能够很好地解决这个问题了。能够复用的关键就是客户端和服务端能够保持长连接，并让一个或者多个连接复用。怎么保持长连接呢？

在BridgeInterceptor的intercept()方法中

requestBuilder.header("Connection", "Keep-Alive")，我们在request的请求头添加了("Connection", "Keep-Alive")的键值对，这样就能够保持长连接。而连接池ConnectionPool就是专门负责管理这些长连接的类。需要注意的是，我们在初始化 ConnectionPool的时候，会设置 闲置的 connections 的最大数量为5个，每个最长的存活时间为5分钟。

b、无缝支持GZIP来减少数据流量

c、缓存响应数据来减少重复的网络请求

d、可以从很多常用的连接问题中自动恢复

2.3.3、OkHttp框架中都用到了哪些设计模式

构造者模式

工厂模式

单例模式

责任链模式等等

8.3 okhttp面试题----拦截器interceptor

相信很多人面试都有遇到这个问题

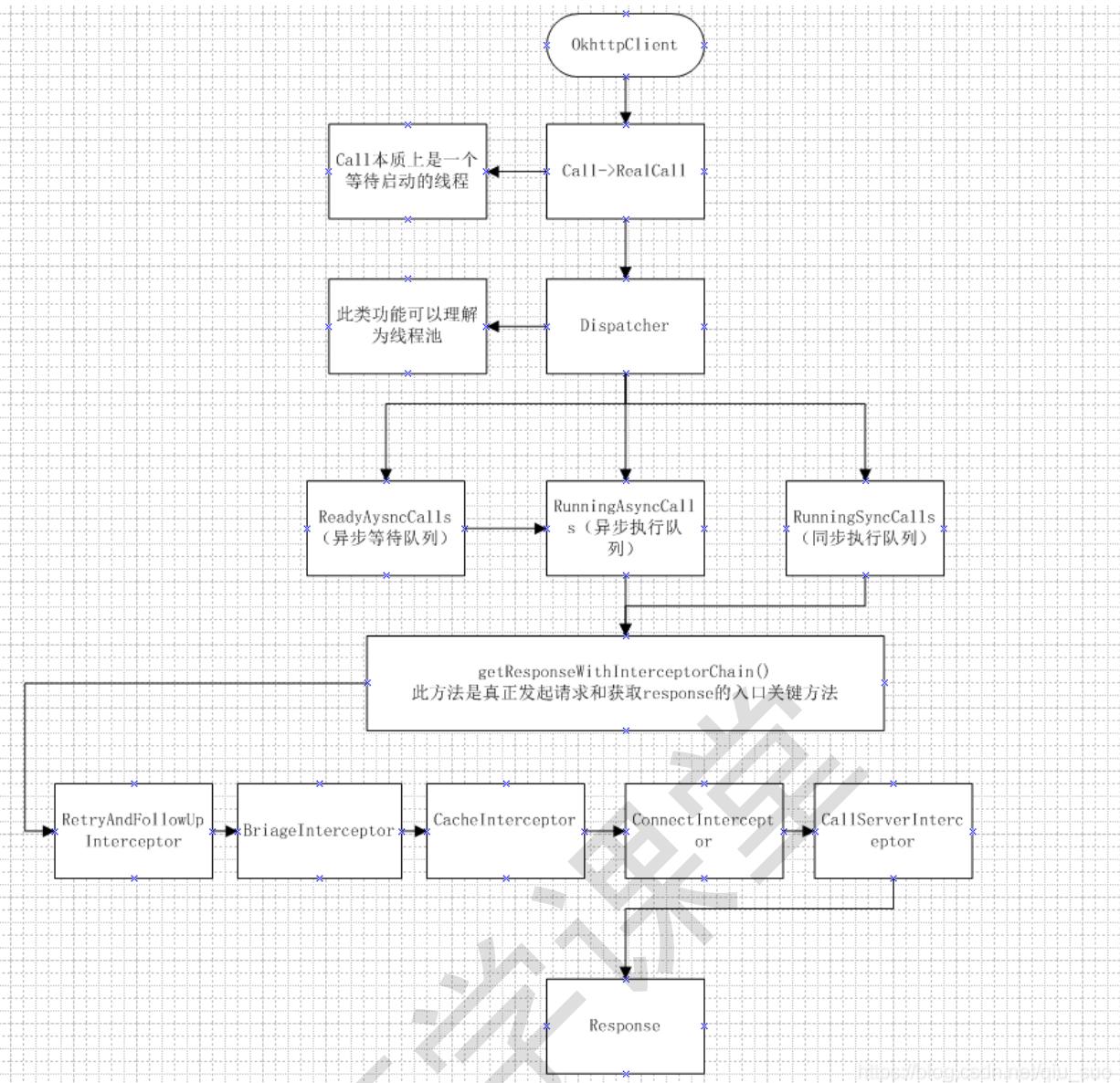
"了解过okhttp吗，简单说一下okhttp的拦截器吧"

我曾经就经历过这样的场景，我只能挠着头说记不清楚了，其实压根没咋细看，面试成不成功的先放一边，面子问题都过不去

回到正题，okhttp在android三方库里面属于一个万金油级别的存在，

1. 首先作为目前最主流的网络请求框架，强是一定的
2. 其次在okhttp中使用了多种设计模式，比如单例，工程，观察者，装饰等经典设计模式，吃透了okhttp的细节也能提升对设计模式的理解
3. okhttp的拦截器各个网络请求过程进行了封装，一方面装了解okhttp的过程中可以加深我们对于http请求的认识，另一方面通过链式结构也方便于我们自定义拦截器进行特定的工作
4. 装okhttp4.0以后，官网源码采用了kotlin语言，没接触kotlin的小伙伴可以借此正好简单接触一下（毕竟官推），也算是让自己多掌握一项技术
5. 如果面试时遇到面试官问了关于okhttp的东西，希望看完这篇文章后的你能跟面试官侃侃而谈，完全讲述okhttp的重要部分怎么也能讲上一二十分钟，将他带进你的节奏吧，okhttp是我的主场

先上一张okhttp的简略流程图



简单的过程是我们通过构建okhttpClient，被转换成一个call也就是请求，然后通过调度器根据启动方式选择加入队列并开启，然后通过五个拦截器补全配置信息并实现网络请求，最终拿到我们所需要的response

拦截器

首先是拦截器的执行顺序

自定义拦截器 - 重定向拦截器 - 桥接（桥梁）连接器 - 缓存拦截器 - 连接拦截器 - 网络拦截器

接下来我会逐一简单讲一下每个拦截器的主要作用和关键代码

RetryAndFollowUpInterceptor - 重定向拦截器

想
識
思
學

```
@Override public Response intercept(Chain chain) throws
IOException { Request request = chain.request();
RealInterceptorChain realChain = (RealInterceptorChain)
chain; Call call = realChain.call(); EventListener
eventListener = realChain.eventListener();
StreamAllocation streamAllocation = new
StreamAllocation(client.connectionPool(),
createAddress(request.url()), call, eventListener,
callStackTrace); this.streamAllocation =
streamAllocation; int followUpCount = 0; Response
priorResponse = null; while (true) { if
(canceled) { streamAllocation.release();
throw new IOException("Canceled"); } Response
response; boolean releaseConnection = true; try
{ response = realChain.proceed(request,
streamAllocation, null, null); releaseConnection =
false; } catch (RouteException e) { // The
attempt to connect via a route failed. The request will
not have been sent. if
(!recover(e.getLastConnectException(), streamAllocation,
false, request)) { throw
e.getFirstConnectException(); } releaseConnection =
false; continue; } catch
(IOException e) { // An attempt to communicate
with a server failed. The request may have been sent.
boolean requestSendStarted = !(e instanceof
ConnectionShutdownException); if (!recover(e,
streamAllocation, requestSendStarted, request)) throw e;
releaseConnection = false; continue; }
finally { // We're throwing an unchecked
exception. Release any resources. if
(releaseConnection) {
streamAllocation.streamFailed(null);
streamAllocation.release(); } } // Attach the prior response if it exists. Such responses
never have a body. if (priorResponse != null) {
response = response.newBuilder()
.priorResponse(priorResponse.newBuilder()
```

```
.body(null) .build())
.build(); } Request followUp; try {
followUp = followUpRequest(response,
streamAllocation.route()); } catch (IOException e) {
    streamAllocation.release(); throw e; }
if (followUp == null) { if (!forWebSocket) {
    streamAllocation.release(); }
return response; }
closeQuietly(response.body()); if (++followUpCount >
MAX_FOLLOW_UPS) { streamAllocation.release();
throw new ProtocolException("Too many follow-up
requests: " + followUpCount); } if
(followUp.body() instanceof UnrepeatableRequestBody) {
    streamAllocation.release(); throw new
HttpRetryException("Cannot retry streamed HTTP body",
response.code()); } if
(!sameConnection(response, followUp.url())) {
    streamAllocation.release(); streamAllocation = new
StreamAllocation(client.connectionPool(),
createAddress(followUp.url()), call, eventListener,
callStackTrace); this.streamAllocation =
streamAllocation; } else if
(streamAllocation.codec() != null) { throw new
IllegalStateException("Closing the body of " + response
+ " didn't close its backing stream. Bad
interceptor?"); } request = followUp;
priorResponse = response; } }
```

interceptor方法是每一个拦截器的核心方法，可以看到的是，在这个重定向拦截器中，我们所填的东西在这里出现了一个streamAllocation，这个类是一些我们http请求的组件和参数，源码量大就不放上来展示了，简单总结一下：

1. 创建streamAllocation 并填充我们请求需要的一部分信息
2. 重连，默认次数为20次，超过则抛出异常，连接成功则将请求和重连结果一并传给下一个拦截器
3. 接收从下一个拦截器传回的response，处理并返回给上一层，也就是我们写的client

BridgeInterceptor - 桥接拦截器

橋接
拦截器

想
識
思
學

```
@Override public Response intercept(Chain chain) throws  
IOException { Request userRequest = chain.request();  
Request.Builder requestBuilder =  
userRequest.newBuilder(); RequestBody body =  
userRequest.body(); if (body != null) { MediaType  
contentType = body.contentType(); if (contentType !=  
null) { requestBuilder.header("Content-Type",  
contentType.toString()); } long contentLength =  
body.contentLength(); if (contentLength != -1) {  
requestBuilder.header("Content-Length",  
Long.toString(contentLength));  
requestBuilder.removeHeader("Transfer-Encoding"); }  
else { requestBuilder.header("Transfer-Encoding",  
"chunked"); requestBuilder.removeHeader("Content-  
Length"); } } if (userRequest.header("Host")  
== null) { requestBuilder.header("Host",  
hostHeader(userRequest.url(), false)); } if  
(userRequest.header("Connection") == null) {  
requestBuilder.header("Connection", "Keep-Alive"); }  
// If we add an "Accept-Encoding: gzip" header field  
we're responsible for also decompressing // the  
transfer stream. boolean transparentGzip = false;  
if (userRequest.header("Accept-Encoding") == null &&  
userRequest.header("Range") == null) {  
transparentGzip = true;  
requestBuilder.header("Accept-Encoding", "gzip"); }  
List<Cookie> cookies =  
cookieJar.loadForRequest(userRequest.url()); if  
(!cookies.isEmpty()) {  
requestBuilder.header("Cookie", cookieHeader(cookies)); }  
if (userRequest.header("User-Agent") == null) {  
requestBuilder.header("User-Agent", version.userAgent()); }  
Response networkResponse =  
chain.proceed(requestBuilder.build());  
HttpHeaders.receiveHeaders(cookieJar, userRequest.url(),  
networkResponse.headers()); Response.Builder  
responseBuilder = networkResponse.newBuilder()  
.request(userRequest); if (transparentGzip &&
```

```
"gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))      &&
HttpHeaders.hasBody(networkResponse)) {           GzipSource
responseBody = new
GzipSource(networkResponse.body().source());       Headers
strippedHeaders = networkResponse.headers().newBuilder()
    .removeAll("Content-Encoding")
    .removeAll("Content-Length")           .build();
responseBuilder.headers(strippedHeaders);       String
contentType = networkResponse.header("Content-Type");
responseBuilder.body(new RealResponseBody(contentType,
-1L, okio.buffer(responseBody))); }     return
responseBuilder.build(); }
```

桥接拦截器更像是一个小小的分层，从源码中不难看出，在这一层中对我们所构建的request的请求头和请求体进行了设置，使得我们所需要发送的请求进一步完善

- 1.对请求头和请求体进行了配置参数的补充并将发起网络请求
- 2.对下一层适配器返回的response进行解压（gzip，简单来说就是将网络请求的数据解压成我们所需要的格式并塞进response的body里）处理并返回给上一层拦截器

CacheInterceptor - 缓存拦截器

```
@Override public Response intercept(Chain chain) throws
IOException {
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

    CacheStrategy strategy = new
    CacheStrategy.Factory(now, chain.request(),
    cacheCandidate).get();

    Request networkRequest = strategy.networkRequest;
```

```
Response cacheResponse = strategy.cacheResponse;

if (cache != null) {
    cache.trackResponse(strategy);
}

if (cacheCandidate != null && cacheResponse == null)
{
    closeQuietly(cacheCandidate.body()); // The cache
candidate wasn't applicable. Close it.
}

// If we're forbidden from using the network and the
cache is insufficient, fail.
if (networkRequest == null && cacheResponse == null)
{
    return new Response.Builder()
        .request(chain.request())
        .protocol(Protocol.HTTP_1_1)
        .code(504)
        .message("Unsatisfiable Request (only-if-
cached)")
        .body(Util.EMPTY_RESPONSE)
        .sentRequestAtMillis(-1L)

    .receivedResponseAtMillis(System.currentTimeMillis())
        .build();
}

// If we don't need the network, we're done.
if (networkRequest == null) {
    return cacheResponse.newBuilder()
        .cacheResponse(stripBody(cacheResponse))
        .build();
}

Response networkResponse = null;
try {
```

```
        networkResponse = chain.proceed(networkRequest);
    } finally {
        // If we're crashing on I/O or otherwise, don't
        // leak the cache body.
        if (networkResponse == null && cacheCandidate != null) {
            closeQuietly(cacheCandidate.body());
        }
    }

    // If we have a cache response too, then we're doing
    // a conditional get.
    if (cacheResponse != null) {
        if (networkResponse.code() == HTTP_NOT_MODIFIED) {
            Response response = cacheResponse.newBuilder()
                .headers(combine(cacheResponse.headers(),
                    networkResponse.headers()))
                .sentRequestAtMillis(networkResponse.sentRequestAtMillis())
                .receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())
                .cacheResponse(stripBody(cacheResponse))
                .networkResponse(stripBody(networkResponse))
                .build();
            networkResponse.body().close();

            // Update the cache after combining headers but
            // before stripping the
            // Content-Encoding header (as performed by
            initContentStream()).
            cache.trackConditionalCacheHit();
            cache.update(cacheResponse, response);
            return response;
        } else {
            closeQuietly(cacheResponse.body());
        }
    }
}
```

```
}

Response response = networkResponse.newBuilder()
    .cacheResponse(stripBody(cacheResponse))
    .networkResponse(stripBody(networkResponse))
    .build();

if (cache != null) {
    if (HttpHeaders.hasBody(response) &&
        CacheStrategy.isCacheable(response, networkRequest)) {
        // Offer this request to the cache.
        CacheRequest cacheRequest = cache.put(response);
        return cachewritingResponse(cacheRequest,
            response);
    }

    if
        (HttpMethod.invalidateCache(networkRequest.method())) {
            try {
                cache.remove(networkRequest);
            } catch (IOException ignored) {
                // The cache cannot be written.
            }
        }
    }
}

return response;
}
```

此部分的代码量较大，其实我想只截主要功能代码，但是想想还是全都放上来了，毕竟这个方法是拦截器最重要的方法，还是都看一下吧，这个拦截器的主要功能就和他的名字一样，主要是关于我们请求的缓存部分

1.cache若不为空则赋cacheCandidate对象

2.获取缓存策略，可以自己设置，默认为

CacheControl.FORCE_NETWORK (即强制使用网络请求)

CacheControl.FORCE_CACHE (即强制使用本地缓存，如果无可用缓存

则返回一个code为504的响应)

3.在不为空且有缓存策略时，若返回304则直接响应缓存创建response返回

4.若无网络或无缓存时返回504

5.在有缓存策略无缓存时调用cache的put方法进行缓存**

ConnectInterceptor - 连接拦截器

interceptor方法

```
@Override public Response intercept(Chain chain) throws  
IOException {    RealInterceptorChain realChain =  
(RealInterceptorChain) chain;    Request request =  
realChain.request();    StreamAllocation streamAllocation  
= realChain.streamAllocation();    // we need the network  
to satisfy this request. Possibly for validating a  
conditional GET.    boolean doExtensiveHealthChecks =  
!request.method().equals("GET");    HttpCodec httpCodec =  
streamAllocation.newStream(client, chain,  
doExtensiveHealthChecks);    RealConnection connection =  
streamAllocation.connection();    return  
realChain.proceed(request, streamAllocation, httpCodec,  
connection); }
```

newsteam方法

```
public HttpCodec newStream(okHttpClient client, boolean doExtensiveHealthChecks) {    int connectTimeout = client.connectTimeoutMillis();    int readTimeout = client.readTimeoutMillis();    int writeTimeout = client.writeTimeoutMillis();    boolean connectionRetryEnabled = client.retryOnConnectionFailure();    try {        RealConnection resultConnection = findHealthyConnection(connectTimeout, readTimeout, writeTimeout, connectionRetryEnabled, doExtensiveHealthChecks);        HttpCodec resultCodec;        if (resultConnection.http2Connection != null) {            resultCodec = new Http2Codec(client, this, resultConnection.http2Connection);        } else {            resultConnection.socket().setSoTimeout(readTimeout);            resultConnection.source.timeout().timeout(readTimeout, MILLISECONDS);            resultConnection.sink.timeout().timeout(writeTimeout, MILLISECONDS);        }        resultCodec = new Http1Codec(            client, this, resultConnection.source, resultConnection.sink);    }    synchronized (connectionPool) {        codec = resultCodec;    }    return resultCodec; } catch (IOException e) {    throw new RouteException(e); }}
```

连接拦截器的工作比较简单，也可以理解为在这一步，真正的网络请求才刚刚开始，之前的部分都属于准备工作，为请求填满所需要的配置信息，在这里拿到了从第一个拦截器就创建的streamAllocation，并生成了io，最后return的proceed的方法，这个方法是来自RealInterceptorChain(翻译：真实拦截器链)的一个方法，其返回值也是response，主要是发起网络请求的重要开端，代码不少，有兴趣的可以自己导个okhttp依赖点进去看一下，总结：

- 1.将url所在的StreamAllocation拿到并生成流 (RealConnect)
- 2.将流和httpcode一并交给下一层拦截器进行请求，并返回response

CallServerInterceptor - 网络拦截器

网络
拦截
器

想
識
思
學

```
public final class CallServerInterceptor implements Interceptor { private final boolean forWebSocket; public CallServerInterceptor(boolean forWebSocket) { this.forWebSocket = forWebSocket; } @Override public Response intercept(Chain chain) throws IOException { HttpCodec httpCodec = ((RealInterceptorChain) chain).httpStream(); StreamAllocation streamAllocation = ((RealInterceptorChain) chain).streamAllocation(); Request request = chain.request(); long sentRequestMillis = System.currentTimeMillis();  
2 httpCodec.writeRequestHeaders(request);  
3 if (HttpMethod.permitsRequestBody(request.method()) && request.body() != null) { Sink requestBodyOut = httpCodec.createRequestBody(request, request.body().contentLength());  
4 BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOut); request.body().writeTo(bufferedRequestBody); bufferedRequestBody.close(); }  
5 httpCodec.finishRequest(); Response response = httpCodec.readResponseHeaders()  
6 .request(request)  
  
.handshake(streamAllocation.connection().handshake())  
7 .sentRequestAtMillis(sentRequestMillis)  
.receivedResponseAtMillis(System.currentTimeMillis())  
8 .build(); int code = response.code(); if (forWebSocket && code == 101) { // Connection is upgrading, but we need to ensure interceptors see a non-null response body. response = response.newBuilder()  
9 .body(Util.EMPTY_RESPONSE) .build(); }  
10 else { response = response.newBuilder()  
11 .body(httpCodec.openResponseBody(response))  
12 .build(); }  
13 if ("close".equalsIgnoreCase(response.request().header("Connection")) || "close".equalsIgnoreCase(response.header("Connection")))  
14 { streamAllocation.noNewStreams(); } if ((code == 204 || code == 205) && response.body().contentLength() > 0) { throw new ProtocolException("HTTP ")  
15 }
```

```
+ code + " had non-zero Content-Length: " +
response.body().contentLength());      }      return
response;  }}
```

最后一个拦截器也是最后真正的请求，十年磨一剑，这个拦截器主要的功能就是发起请求和拿到并处理请求结果

- 1.发起请求**
- 2.完成读写**
- 3.根据返回码处理请求结果**
- 4.关闭连接**

```
\=====
=====
```

以上便是五个拦截器以及他们的主要功能，源码量很大，我只截了比较重要的部分，对于我自己而言我也不敢保证自己吃透了okhttp，还有一些部分理解的不够深刻，有不对的地方也请指正，相互学习讨论

关于自定义拦截器，我手上没有现存的demo，简单的去说继承
interceptor类在里面怎样怎样谁都会说，所以我就不耽误大家了，毕竟
网上也有很多大神写的自定义拦截器连源码带解析整的比我好太多，我没
写不代表不重要，面试中也是很重要的一环，希望大家也再搜一下一定要
看全面了

```
\=====
=====
```

我们刚才依次了解了五个拦截器的功能，接下来看一下他们的存在方式
和调用

在RealCall这个类里面有一行代码十分重要

```
Response response = getResponseWithInterceptorChain();
```

这行代码也是设置拦截器的入口，也是拦截器依次链式执行的开始
让我们来看一下这个方法里有什么

```
Response getResponseWithInterceptorChain() throws  
IOException { // Build a full stack of interceptors.  
List<Interceptor> interceptors = new ArrayList<>();  
interceptors.addAll(client.interceptors());  
interceptors.add(retryAndFollowUpInterceptor);  
interceptors.add(new  
BridgeInterceptor(client.cookieJar()));  
interceptors.add(new  
CacheInterceptor(client.internalCache()));  
interceptors.add(new ConnectInterceptor(client)); if  
(!forWebSocket) {  
interceptors.addAll(client.networkInterceptors()); }  
interceptors.add(new  
CallServerInterceptor(forWebSocket));  
Interceptor.Chain chain = new  
RealInterceptorChain(interceptors, null, null, null, 0,  
originalRequest, this, eventListener,  
client.connectTimeoutMillis(),  
client.readTimeoutMillis(), client.writeTimeoutMillis());  
return chain.proceed(originalRequest); }
```

不难看出，首先创建了一个集合用来存储适配器，，第一个
addAll(client.interceptors())便是添加了我们的自定义适配器，后面是五
个适配器依次加入，最终生成了一个Chain，也就是拦截器链
在这个方法的最后，调用了chain的proceed方法，不管是从英文水平上
来说还是用脚指头想，这个方法肯定是要遍历轮询拦截器让他们干活了，
我们再来看一下proceed方法

```
public Response proceed(Request request, StreamAllocation streamAllocation, HttpCodec httpCodec, RealConnection connection) throws IOException {    if (index >= interceptors.size()) throw new AssertionError();    calls++; // If we already have a stream, confirm that the incoming request will use it.    if (this.httpCodec != null && !this.connection.supportsUrl(request.url())) {        throw new IllegalStateException("network interceptor " + interceptors.get(index - 1) + " must retain the same host and port");    } // If we already have a stream, confirm that this is the only call to chain.proceed().    if (this.httpCodec != null && calls > 1) {        throw new IllegalStateException("network interceptor " + interceptors.get(index - 1) + " must call proceed() exactly once");    } // Call the next interceptor in the chain.    RealInterceptorChain next = new RealInterceptorChain(interceptors, streamAllocation, httpCodec, connection, index + 1, request);    Interceptor interceptor = interceptors.get(index);    Response response = interceptor.intercept(next); // Confirm that the next interceptor made its required call to chain.proceed().    if (httpCodec != null && index + 1 < interceptors.size() && next.calls != 1) {        throw new IllegalStateException("network interceptor " + interceptor + " must call proceed() exactly once");    } // Confirm that the intercepted response isn't null.    if (response == null) {        throw new NullPointerException("interceptor " + interceptor + " returned null");    }    return response; }
```

我们在这一堆判断和处理挑出了两行行亮眼而又亮眼的代码

**Interceptor interceptor = interceptors.get(index);
Response response = interceptor.intercept(next);**

不是说client的一连串点方法叫链式结构，在这里链式结构被解释的淋漓尽致，通过获取下标的方式执行当前拦截器的interceptor方法并从其下一个拦截器中拿到返回回来的response，至此调用和轮询部分也结束了

再来说说response颠沛流离的一生

首先次对象在重定向拦截器总被创建但赋值为null

在经历缓存拦截器时会根据缓存策略，若缓存中存在则直接生成response并返回，若没有则继续向下

在网络拦截器中被请求并真正意义上的赋值，但请求的到的是io写下来的网络字节

然后response通过一层层拦截器传回到桥接拦截器进行gzip解压，这也是为什么桥接拦截器在缓存拦截器之前的原因之一（gzip不了解的小伙伴建议百度一下，多懂一些知识总没坏处）

最终经历了一整个拦截器链得到的一个带有我们所需要信息的response，我们再写一个线程调度将请求结果返给ui线程进行使用

关于okhttp的dispatcher部分后面我会再详细的写一篇，为什么不在这篇中一起讲解了呢？

原因简单而现实 因为我快下班了hhhhhhhhhhhhhh

当然我也会尽快抽时间将dispatcher的部分也做出来，讲当然是讲全套的，相信有耐心一点点看到这里的人个个都是彭于晏胡歌吴彦祖吧，说不定里面可能还混着一两个范冰冰杨幂

那今天就先到这里啦

INTERESTING!!!

8.4 Okhttp3 总结研究（面试）

OKhttp3 是最近比较主流的网络请求框架。面试中，常会问道你对 okhttp3是否有深入的了解。在这篇文章我总结了下okhttp3的原理（非用法，用法自己百度就行），以及大神们根据源码分析OKhttp3比较好的文章。

8.4.1首先是大神们对Okhttp3的源码分析：(转载)

1.1 Android OkHttp (三) 源码解析

(这是我第一篇看的okhttp的源码解析，根据这篇文章和自己查看okhttp 源码，看okhttp内部，这篇文章大概讲清楚okhttp 内部构造，虽然有些语言表达上的错误，这篇也是针对okhttp的。建议可以先看这篇，先了解大概)

前面两篇文章，我们介绍了OKHttp的基本用法，今天这篇文章将从OkHttpClient的角度来分析OkHttp的整个工作流程。

初始化一个OkHttpClient对象，

```
OkHttpClient mOkHttpClient = new OkHttpClient();
```

OkHttpClient的构造函数中又初始化了一个Builder对象，

```
public OkHttpClient() {  
    this(new Builder());  
}
```

Builder的构造函数中，默认会创建对象以及一些属性，Builder对象是用来构造网络请求的。这种设计模式和Android的对话框中的Builder几乎是一个意思。

```
public Builder() {      dispatcher = new Dispatcher();  
    protocols = DEFAULT_PROTOCOLS;      connectionSpecs =  
DEFAULT_CONNECTION_SPECS;      proxySelector =  
ProxySelector.getDefault();      cookieJar =  
CookieJar.NO_COOKIES;      socketFactory =  
SocketFactory.getDefault();      hostnameVerifier =  
OkHostnameVerifier.INSTANCE;      certificatePinner =  
CertificatePinner.DEFAULT;      proxyAuthenticator =  
Authenticator.NONE;      authenticator =  
Authenticator.NONE;      connectionPool = new  
ConnectionPool();      dns = Dns.SYSTEM;  
followSslRedirects = true;      followRedirects = true;  
retryOnConnectionFailure = true;      connectTimeout =  
10_000;      readTimeout = 10_000;      writeTimeout =  
10_000;  }
```

接着，创建一个请求，

```
Request request = new Request.Builder()  
    .url(url)  
    .build();
```

Request类是HTTP请求，它携带了请求地址、请求方法、请求头部、请求体以及其他信息。它也是通过Builder模式创建的。

Request.Builder().build()源码，

```
public Request build() {      if (url == null) throw new  
IllegalStateException("url == null");      return new  
Request(this);  }
```

Request的构造方法，

```
private Request(Builder builder) {      this.url =  
builder.url;      this.method = builder.method;  
this.headers = builder.headers.build();      this.body =  
builder.body;      this.tag = builder.tag != null ?  
builder.tag : this;  }
```

发起请求有两种方式，一种是同步，一种是异步，下面分别看看如何实现，

同步请求，

```
Response response =  
    mokHttpClient.newCall(request).execute();
```

Response是HTTP响应，它继承自Closeable（Closeable继承自AutoCloseable，AutoCloseable资源自动关闭的接口），它携带了请求、网络请求协议、返回状态码、请求头、响应体等等，其中，网络请求协议是Protocol，Protocol是一个枚举类型，包含4中类型，

```
public enum Protocol { /** * An obsolete plaintext framing that does not use persistent sockets by default. */
    HTTP_1_0("http/1.0"), /** * A plaintext framing that includes persistent connections. */
    

This version of OkHttp implements <a href="http://www.ietf.org/rfc/rfc2616.txt">RFC 2616</a>, and tracks revisions to that spec.


    HTTP_1_1("http/1.1"), /** * Chromium's binary-framed protocol that includes header compression, multiplexing multiple requests on the same socket, and server-push. HTTP/1.1 semantics are layered on SPDY/3. */
    

This version of OkHttp implements SPDY 3 <a href="http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1">draft 3.1</a>. Future releases of OkHttp may use this identifier for a newer draft of the SPDY spec.


    SPDY_3("spdy/3.1"), /** * The IETF's binary-framed protocol that includes header compression, multiplexing multiple requests on the same socket, and server-push. HTTP/1.1 semantics are layered on HTTP/2. */
    

HTTP/2 requires deployments of HTTP/2 that use TLS 1.2 support


    {@linkplain
        Ciphersuite#TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256} ,
        present in Java 8+ and Android 5+. Servers that enforce this may send an exception message including the string {@code * INADEQUATE_SECURITY}.
    }
    HTTP_2("h2");
}
```

接着看

```
OkHttpClient.newCall(request);
```

Call类是一个准备执行的请求，它可以被取消，它代表着一个单一的请求/响应流，不能被执行两次。Call类是一个接口，

```
public interface Call { /** Returns the original request  
that initiated this call. */ Request request(); /** *  
Invokes the request immediately, and blocks until the  
response can be processed or is in * error. * *  
<p>The caller may read the response body with the  
response's {@link Response#body} method. To * avoid  
leaking resources callers must {@linkplain ResponseBody  
close the response body}. * * <p>Note that  
transport-layer success (receiving a HTTP response code,  
headers and body) does * not necessarily indicate  
application-layer success: {@code response} may still  
indicate an * unhappy HTTP response code like 404 or  
500. * * @throws IOException if the request could  
not be executed due to cancellation, a connectivity *  
problem or timeout. Because networks can fail during an  
exchange, it is possible that the * remote server  
accepted the request before the failure. * * @throws  
IllegalStateException when the call has already been  
executed. */ Response execute() throws  
IOException; /** * Schedules the request to be  
executed at some point in the future. * * <p>The  
{@link OkHttpClient#dispatcher dispatcher} defines when  
the request will run: usually * immediately unless  
there are several other requests currently being  
executed. * * <p>This client will later call back  
{@code responseCallback} with either an HTTP response or  
a * failure exception. * * @throws  
IllegalStateException when the call has already been  
executed. */ void enqueue(Callback  
responseCallback); /** Cancels the request, if possible.  
Requests that are already complete cannot be canceled. */  
void cancel(); /** * Returns true if this call has  
been either {@linkplain #execute() executed} or  
{@linkplain * #enqueue(Callback) enqueued}. It is an  
error to execute a call more than once. */  
boolean isExecuted(); boolean isCanceled(); interface  
Factory { Call newCall(Request request); }}
```

newCall()方法是创建一个Call对象，

```
@Override     public Call newCall(Request request) {  
    return new RealCall(this, request); }
```

RealCall类实现了Call类，下面展示RealCall()方法的代码，

```
protected RealCall(okHttpClient client, Request  
originalRequest) {    this.client = client;  
this.originalRequest = originalRequest;  
this.retryAndFollowUpInterceptor = new  
RetryAndFollowUpInterceptor(client); }
```

接着是RealCall的execute()方法。execute()方法是同步方法，即一直等待http请求，直到返回了响应。在这之间会阻塞进程，所以通过同步方法不能在Android的主线程中执行，否则会报错。

OKHttp提供了execute()方法（同步方法）和enqueue()方法（异步方法），下面我们先看看execute()方法（同步方法），

```
@Override public Response execute() throws IOException {  
    synchronized (this) {        if (executed) throw new  
IllegalStateException("Already Executed");        executed  
= true;    }    try {  
        client.dispatcher().executed(this);        Response result  
= getResponseWithInterceptorChain();        if (result ==  
null) throw new IOException("Canceled");        return  
result;    } finally {  
        client.dispatcher().finished(this);    } }
```

execute()方法，首先判断是否已执行过，如果已经执行过，则抛出异常信息，也就是说一次Call实例只能调用一次execute()方法，和我们之前说的一样。

如果未执行，则调用Dispatcher类的executed()方法将该Call加入到一个双端队列中，

```
...     private final Deque<RealCall> runningSyncCalls =  
new ArrayDeque<>(); //双端队列 ...      /** Used by  
{@code Call#execute} to signal it is in-flight. */  
synchronized void executed(RealCall call) {  
runningSyncCalls.add(call); }
```

接着调用getResponseWithInterceptorChain()方法返回Response对象，最后finally中，调用Dispatcher的finished()方法，在从已经执行的双端队列中移除本次Call。

```
void finished(RealCall call) {  
finished(runningSyncCalls, call, false); } private <T>  
void finished(Deque<T> calls, T call, boolean  
promoteCalls) { int runningCallsCount; Runnable  
idleCallback; synchronized (this) { if  
(!calls.remove(call)) throw new AssertionError("Call  
wasn't in-flight!"); if (promoteCalls)  
promoteCalls(); runningCallsCount =  
runningCallsCount(); idleCallback =  
this.idleCallback; } if (runningCallsCount == 0 &&  
idleCallback != null) { idleCallback.run(); } }
```

通过上面分析，真正发出网络请求，返回结果应该是getResponseWithInterceptorChain()方法，那么接下来，主要看看这个方法

```
private Response getResponseWithInterceptorChain() throws  
IOException { // Build a full stack of interceptors.  
List<Interceptor> interceptors = new ArrayList<>();  
interceptors.addAll(client.interceptors());  
interceptors.add(retryAndFollowUpInterceptor);  
interceptors.add(new  
BridgeInterceptor(client.cookieJar()));  
interceptors.add(new  
CacheInterceptor(client.internalCache()));  
interceptors.add(new ConnectInterceptor(client)); if  
(!retryAndFollowUpInterceptor.isForWebSocket()) {  
interceptors.addAll(client.networkInterceptors()); }  
interceptors.add(new CallServerInterceptor(  
retryAndFollowUpInterceptor.isForWebSocket()));  
Interceptor.Chain chain = new RealInterceptorChain(  
interceptors, null, null, null, 0, originalRequest);  
return chain.proceed(originalRequest); }
```

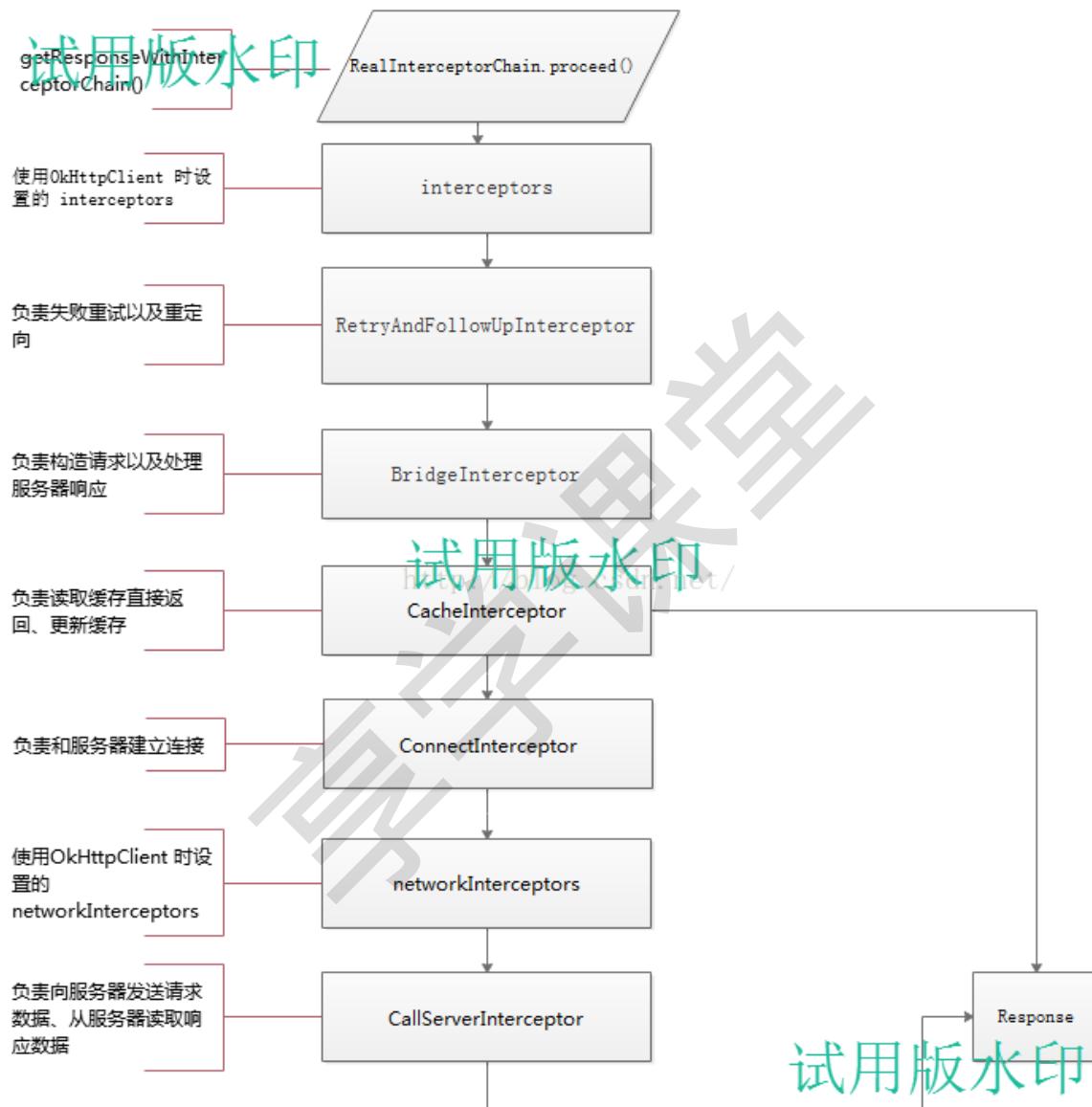
仔细查看便会发现，该方法是组装各种拦截器为一个拦截器链，最后调用RealInterceptorChain的proceed()方法，来处理这个请求，

```
@Override public Response proceed(Request request) throws  
IOException { return proceed(request,  
streamAllocation, httpStream, connection); } public  
Response proceed(Request request, StreamAllocation  
streamAllocation, HttpStream httpStream, Connection  
connection) throws IOException { if (index >=  
interceptors.size()) throw new AssertionError();  
calls++; // If we already have a stream, confirm  
that the incoming request will use it. if  
(this.httpStream != null &&  
!sameConnection(request.url())) { throw new  
IllegalStateException("network interceptor " +  
interceptors.get(index - 1) + " must retain the  
same host and port"); } // If we already have  
a stream, confirm that this is the only call to  
chain.proceed(). if (this.httpStream != null && calls  
> 1) { throw new IllegalStateException("network  
interceptor " + interceptors.get(index - 1) + "  
must call proceed() exactly once"); } // Call  
the next interceptor in the chain.  
RealInterceptorChain next = new RealInterceptorChain(  
interceptors, streamAllocation, httpStream,  
connection, index + 1, request); Interceptor  
interceptor = interceptors.get(index); Response  
response = interceptor.intercept(next); //  
Confirm that the next interceptor made its required call  
to chain.proceed(). if (httpStream != null && index +  
1 < interceptors.size() && next.calls != 1) { throw  
new IllegalStateException("network interceptor " +  
interceptor + " must call proceed() exactly  
once"); } // Confirm that the intercepted  
response isn't null. if (response == null) {  
throw new NullPointerException("interceptor " +  
interceptor + " returned null"); } return  
response; }
```

拦截器Interceptor和拦截器链Chain都是接口，

```
public interface Interceptor { Response intercept(Chain chain) throws IOException; interface Chain { Request request(); Response proceed(Request request) throws IOException; Connection connection(); } }
```

下面用一张流程图来说明拦截器链递归从拦截器中返回Response（这个过程很类似Android中的事件处理机制）



下面再看看enqueue()方法（异步方法），指在另外的工作线程中执行http请求，请求时不会阻塞当前的线程，所以可以在Android主线程中使用。

```
@Override public void enqueue(CallBack responseCallback)
{    synchronized (this) {        if (executed) throw new
IllegalStateException("Already Executed");        executed
= true;    }    client.dispatcher().enqueue(new
AsyncCall(responseCallback)); }
```

该方法和同步方法一样，首先都校验这个Call是否已经被执行，如果执行过，就报异常。如果未执行，则调用Dispatcher分发器的enqueue()方法。首先，我们看看AsyncCall这个类，

```
final class AsyncCall extends NamedRunnable {    private
final Callback responseCallback;    private
AsyncCall(Callback responseCallback) {        super("okHttp
%s", redactedUrl().toString());
this.responseCallback = responseCallback;    }
String host() {        return originalRequest.url().host();
}    Request request() {        return
originalRequest;    }    RealCall get() {
return RealCall.this;    }    @Override protected
void execute() {        boolean signalledCallback = false;
        try {            Response response =
getResponsewithInterceptorChain();            if
(retryAndFollowUpInterceptor.isCanceled()) {
signalledCallback = true;
responseCallback.onFailure(RealCall.this, new
IOException("Canceled"));            } else {
signalledCallback = true;
responseCallback.onResponse(RealCall.this, response);
}        } catch (IOException e) {            if
(signalledCallback) {                // Do not signal the
callback twice!
Platform.get().log(INFO,
"Callback failure for " + toLoggableString(), e);
} else {
responseCallback.onFailure(RealCall.this, e);            }
} finally {            client.dispatcher().finished(this);
        }    }public abstract class NamedRunnable
implements Runnable {    protected final String name;
public NamedRunnable(String format, Object... args) {
this.name = Util.format(format, args);    }    @Override
public final void run() {        String oldName =
Thread.currentThread().getName();
Thread.currentThread().setName(name);        try {
execute();    } finally {
Thread.currentThread().setName(oldName);    }    }
protected abstract void execute();}
```

AsyncCall-实际上是一个Runnable，在run()方法中调用了execute()方法，在该方法中，我们又看到既熟悉又陌生的代码

```
(getResponsewithInterceptorChain()) , Response response  
= getResponsewithInterceptorChain();
```

执行该方法后，根据响应结果设置回调方法的结果。AsyncCall执行完调用Dispatcher的finished(this)方法，

```
void finished(AsyncCall call) {  
    finished(runningAsyncCalls, call, true); } private <T>  
void finished(Deque<T> calls, T call, boolean  
promoteCalls) { int runningCallsCount; Runnable  
idleCallback; synchronized (this) { if  
(!calls.remove(call)) throw new AssertionError("Call  
wasn't in-flight!"); if (promoteCalls)  
promoteCalls(); runningCallsCount =  
runningCallsCount(); idleCallback =  
this.idleCallback; } if (runningCallsCount == 0 &&  
idleCallback != null) { idleCallback.run(); } }
```

finished()方法先从runningAsyncCalls(异步请求队列)删除已经执行的异步请求，然后接着调用了promoteCalls()方法，

```
private void promoteCalls() { if  
(runningAsyncCalls.size() >= maxRequests) return; //  
Already running max capacity. if  
(readyAsyncCalls.isEmpty()) return; // No ready calls to  
promote. for (Iterator<AsyncCall> i =  
readyAsyncCalls.iterator(); i.hasNext(); ) {  
    AsyncCall call = i.next(); if  
(runningCallsForHost(call) < maxRequestsPerHost) {  
        i.remove(); runningAsyncCalls.add(call);  
        executorService().execute(call); } if  
(runningAsyncCalls.size() >= maxRequests) return; //  
Reached max capacity. } }
```

首先判断runningAsyncCalls(异步请求队列)是否还有请求，如果有，则返回；否则readyAsyncCalls (异步调用准备任务) 是否为空，如果为空，则返回；否则，循环readyAsyncCalls (异步调用准备任务)，将call加入到runningAsyncCalls(异步请求队列)中，并在readyAsyncCalls (异步调

用准备任务) 删除掉该call, 接着线程池执行call。

下面接着看Dispatcher分发器的enqueue()方法,

```
synchronized void enqueue(AsyncCall call) {    if  
    (runningAsyncCalls.size() < maxRequests &&  
    runningCallsForHost(call) < maxRequestsPerHost) {  
    runningAsyncCalls.add(call);  
    executorService().execute(call);    } else {  
    readyAsyncCalls.add(call);    } }
```

如果runningAsyncCalls的大小小于最大请求数量 (最大线程数量、并发数量) 并且call小于最大主机请求限制, 那么将call 加入到runningAsyncCalls中, 接着线程池执行call; 否则, 将call加入到readyAsyncCalls (异步调用准备任务) 。

PS: runningCallsForHost()方法, 循环判断cal的host和runningAsyncCalls的中的call的host相同的数据。同一请求是否超过想过请求同时存在的最大值。

```
private int runningCallsForHost(AsyncCall call) {    int  
result = 0;    for (AsyncCall c : runningAsyncCalls) {  
    if (c.host().equals(call.host())) result++;    }  
return result; }
```

可以看到请求的最核心代码, 与Dispatcher类分不开, 下面就看看Dispatcher类, Dispatcher是异步请求的策略,

```
public final class Dispatcher { private int maxRequests  
= 64; private int maxRequestsPerHost = 5; private  
Runnable idleCallback; /** Executes calls. Created  
lazily. */ private ExecutorService executorService; /**  
Ready async calls in the order they'll be run. */  
private final Deque<AsyncCall> readyAsyncCalls = new  
ArrayDeque<>(); /** Running asynchronous calls. Includes  
canceled calls that haven't finished yet. */ private  
final Deque<AsyncCall> runningAsyncCalls = new  
ArrayDeque<>(); /** Running synchronous calls. Includes  
canceled calls that haven't finished yet. */ private  
final Deque<RealCall> runningSyncCalls = new ArrayDeque<>  
(); ...}
```

Dispatcher维护了如下变量，

int maxRequests = 64: 最大并发请求数为64

int maxRequestsPerHost = 5: 每个主机最大请求数为5

Runnable idleCallback: Runnable对象，在删除任务时执行

ExecutorService executorService: 消费者池（也就是线程池）

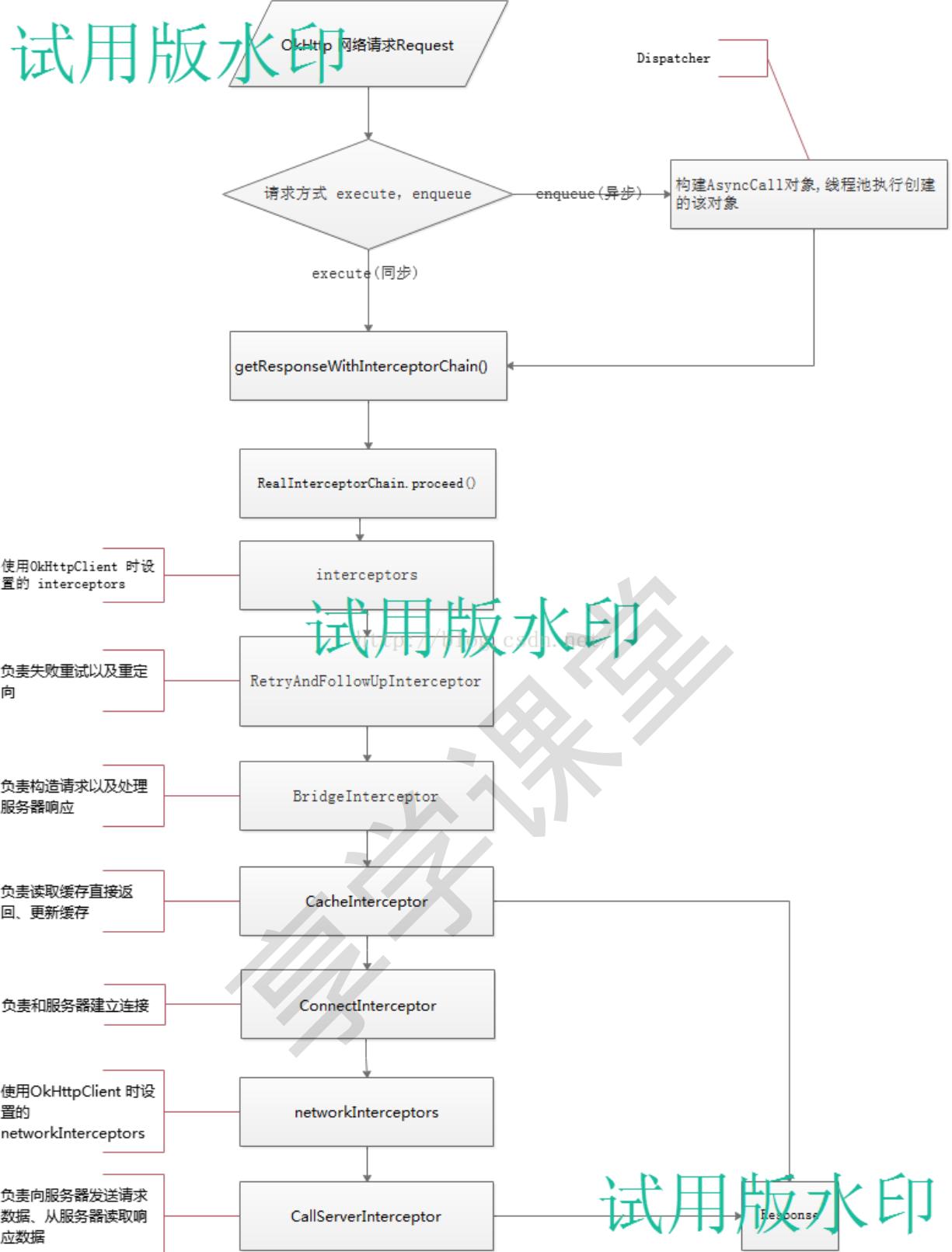
Deque readyAsyncCalls: 缓存，异步调用准备执行任务

Deque runningAsyncCalls: 正在运行的异步任务，包含取消尚未完成的调用

Deque runningSyncCalls: 正在运行的同步任务，包含取消尚未完成的调用

至此，OKHttp的源码分析就结束了！希望本篇文章对读者有所收获！

下面结合本篇文章，总结OKHttp的整体流程，流程图如下所示，



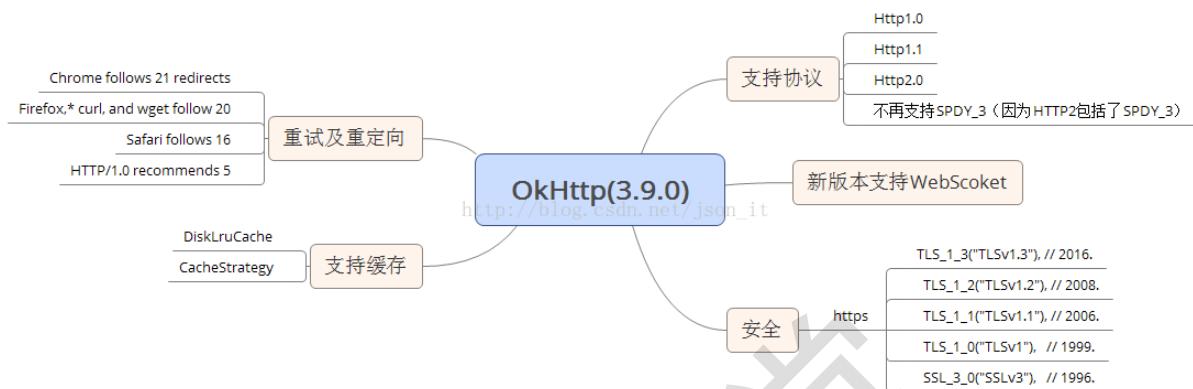
8.4.2 okhttp源码解析

(这是第二篇，也是最近在看的一篇，大神分析的比较全，也比较准确，
详细解析的okhttp3)

OkHttp是一个非常优秀的网络请求框架，已被谷歌加入到Android的源码中。目前比较流行的Retrofit也是默认使用OkHttp的。所以OkHttp的源码是一个不容错过的学习资源，学习源码之前，务必熟练使用这个框架，否则就是跟自己过不去。

use -> running source code -> reading & learning the source code.

1、初识



在早期的版本中，OkHttp支持Http1.0,1.1,SPDY协议，但是Http2协议的问世，导致OkHttp也做出了改变，OkHttp鼓励开发者使用HTTP2，不再对SPDY协议给予支持。另外，新版本的OkHttp还有一个新的亮点就是支持WebScoket，这样我们就可以非常方便的建立长连接了。[关于Http各个版本的异同，可以查看这篇博客：](#)

http://blog.csdn.net/json_it/article/details/78312311

作为一个优秀的网络框架，OkHttp同样支持网络缓存，OkHttp的缓存基于DiskLruCache,对这个类不熟悉的可以[这里学习](#)。DiskLruCache虽然没有被收入到Android的源码中，但也是谷歌推荐的一个优秀的缓存框架。有时间可以自己学习源码，这里不再叙述。

在安全方面，OkHttp目前支持了如上图所示的TLS版本，以确保一个安全的Socket连接。重试及重定向就不再说了，都知道什么意思，左上角给出了各浏览器或Http版本支持的重试或重定向次数。

2、流程（以同步请求为例）

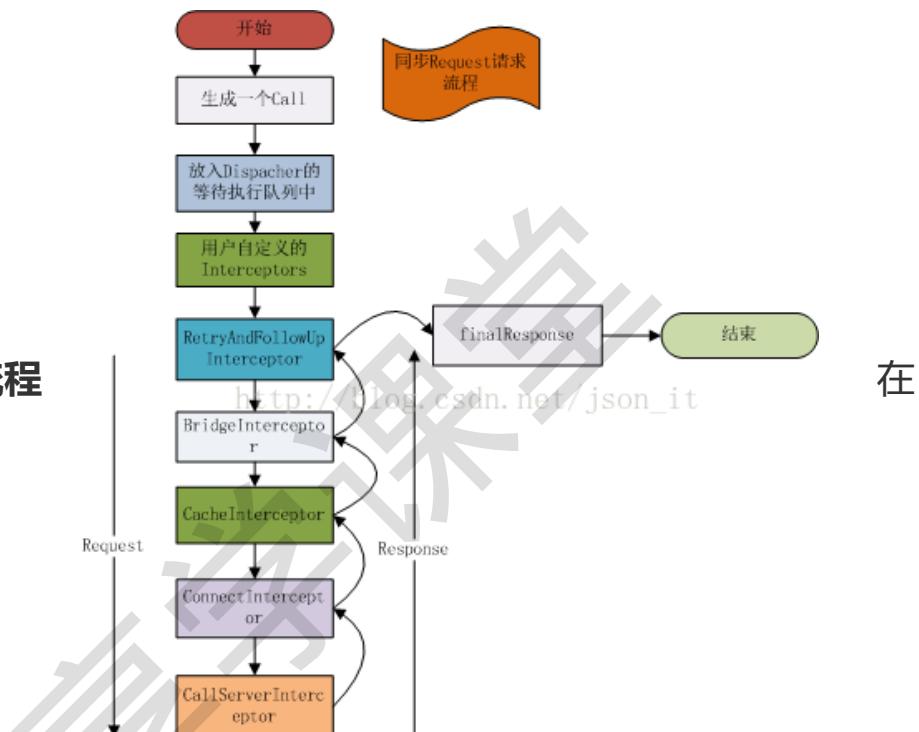
2.1、基本使用

```

OkHttpClient client = new OkHttpClient();
Request request = new
Request.Builder().url("http://www.baidu.com")
    .build();                                try {
Response response = client.newCall(request).execute();
if (response.isSuccessful()) {
    System.out.println("成功");
}
} catch (IOException e) {
e.printStackTrace();
}

```

2.2、同步请求流程



开始流程讲解之前，先了解一下三个概念的含义（以下来自源码注释）：

Connections：连接远程服务器的物理连接；

Streams：基于Connection的逻辑Http请求/响应对。一个连接可以承载多少个Stream都是有限制的，Http1.x连接只能承载一个Stream，而一个Http2.0连接可以承载多个Stream（支持并发请求，并发请求共用一个Connection）；

Calls：逻辑Stream序列，典型的例子是一个初始请求及其后续的请求。We prefer to keep all streams of a single call on the same connection for better behavior and locality.

对于同步和异步请求，唯一的区别就是异步请求会放在线程池（ThreadPoolExecutor）中去执行，而同步请求则会在当前线程中执行，注意：同步请求会阻塞当前线程。

对于Http1.1，call - 1:1 - Stream - 1:1 - connection;

对于http2.0,call - 1:1 - Stream - N:1 - connection;

由上述流程图，我们可以直观的了解到一次基本的请求包括如下两个部分：call+interceptors。call:最终的请求对象；

interceptors:这是OkHttp最核心的部分，一个请求会经过OkHttp的若干个拦截器进行处理，每一个拦截器都会完成一个功能模块，比如 CacheInterceptor完成网络请求的缓存。一个Request经过拦截器链的处理之后，会得到最终的Response。

interceptors里面包括的东西很多东西，后续的源码分析就是以拦截器为主线来进行分析。

3、源码分析

```
OkHttpClient client = new OkHttpClient();
Request request = new
Request.Builder().url("http://www.baidu.com")
    .build();                                try {
    Response response =
client.newCall(request).execute();
if (response.isSuccessful()) {
System.out.println("成功");                }
} catch (IOException e) {
e.printStackTrace();                      }
```

3.1、OkHttpClient

首先，我们生成了一个OKHttpClient对象，注意OKHttpClient对象的生成有两种方式：一种是我们使用的方式，另一种是使用建造者（Builder）模式 -- new OkHttpClient.Builder()....Build()。那么这两种方式有什么区别呢？

第一种：

```
public OkHttpClient() {    this(new Builder()); }
public Builder() {    dispatcher = new Dispatcher();
    protocols = DEFAULT_PROTOCOLS;    connectionSpecs =
DEFAULT_CONNECTION_SPECS;    eventListenerFactory =
EventListener.factory(EventListener.NONE);
proxySelector = ProxySelector.getDefault();
cookieJar = CookieJar.NO_COOKIES;    socketFactory =
SocketFactory.getDefault();    hostnameVerifier =
okHostnameVerifier.INSTANCE;    certificatePinner =
CertificatePinner.DEFAULT;    proxyAuthenticator =
Authenticator.NONE;    authenticator =
Authenticator.NONE;    connectionPool = new
ConnectionPool();    dns = Dns.SYSTEM;
followSslRedirects = true;    followRedirects = true;
    retryOnConnectionFailure = true;    connectTimeout =
10_000;    readTimeout = 10_000;    writeTimeout =
10_000;    pingInterval = 0; }
```

可以看到我们简单的一句new OkHttpClient(), OkHttp就已经为我们做了很多工作，很多我们需要的参数在这里都获得默认值。各字段含义如下：

dispatcher: 直译就是调度器的意思。主要作用是通过双端队列保存Calls (同步&异步Call) , 同时在线程池中执行异步请求。后面会详细解析该类。

protocols: 默认支持的Http协议版本 -- Protocol.HTTP_2, Protocol.HTTP_1_1;

connectionSpecs: OKHttp连接 (Connection) 配置 -- ConnectionSpec.MODERN_TLS, ConnectionSpec.CLEARTEXT, 我们分别看一下:

```
/** TLS 连接 */ public static final ConnectionSpec  
MODERN_TLS = new Builder(true)  
.cipherSuites(APPROVED_CIPHER_SUITES)  
.tlsVersions(TlsVersion.TLS_1_3, TlsVersion.TLS_1_2,  
TlsVersion.TLS_1_1, TlsVersion.TLS_1_0)  
.supportsTlsExtensions(true) .build(); /** 未加密、未  
认证的Http连接. */ public static final ConnectionSpec  
CLEARTEXT = new Builder(false).build();
```

可以看出一个是针对TLS连接的配置，一个是针对普通的Http连接的配置；

eventListenerFactory：一个Call的状态监听器，注意这个是okhttp新添加的功能，目前还不是最终版，在后面的版本中会发生改变的。

proxySelector：使用默认的代理选择器；

cookieJar：默认是没有Cookie的； socketFactory：使用默认的Socket工厂产生Socket；

hostnameVerifier、certificatePinner、proxyAuthenticator、

authenticator：安全相关的设置； connectionPool：连接池；后面会详细介绍；

dns:这个一看就知道，域名解析系统 domain name -> ip address；

pingInterval :这个就和WebSocket有关了。为了保持长连接，我们必须间隔一段时间发送一个ping指令进行保活；

第二种：默认的设置和第一种方式相同，但是我们可以利用建造者模式单独的设置每一个属性； 注意事项：OkHttpClient强烈建议全局单例使用，因为每一个OkHttpClient都有自己单独的连接池和线程池，复用连接池和线程池能够减少延迟、节省内存。

3.2、RealCall (生成一个Call)

在我们定义了请求对象request之后，我们需要生成一个Call对象，该对象代表了一个准备被执行的请求。Call是可以被取消的。Call对象代表了一个request/response 对 (Stream) .还有一个就是Call只能被执行一次。执行同步请求，代码如下 (RealCall的execute方法)：

```
@Override public Response execute() throws IOException
{
    synchronized (this) {
        if (executed) throw new
IllegalStateException("Already Executed");
        executed
= true;
    }
    captureCallStackTrace();
    eventListener.callStart(this);
    try {
client.dispatcher().executed(this);
        Response result
= getResponsewithInterceptorChain();
        if (result ==
null) throw new IOException("Canceled");
        return
result;
    } catch (IOException e) {
eventListener.callFailed(this, e);
        throw e;
    }
finally {
        client.dispatcher().finished(this);
    }
}
```

解析：首先如果executed等于true，说明已经被执行，如果再次调用执行就抛出异常。这说明了一个Call只能被执行。注意此处同步请求与异步请求生成的Call对象的区别，执行异步请求代码如下（RealCall的enqueue方法）：

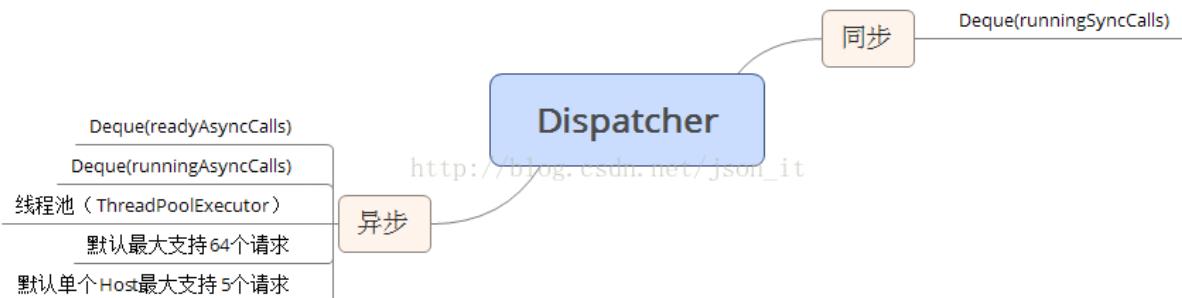
```
@Override public void enqueue(Callback
responseCallback) {
    synchronized (this) {
        if
(executed) throw new
IllegalStateException("Already
Executed");
        executed = true;
    }
    captureCallStackTrace();
    eventListener.callStart(this);
    client.dispatcher().enqueue(new
AsyncCall(responseCallback));
}
```

可以看到同步请求生成的是RealCall对象，而异步请求生成的是AsyncCall对象。AsyncCall说到底其实就是Runnable的子类。

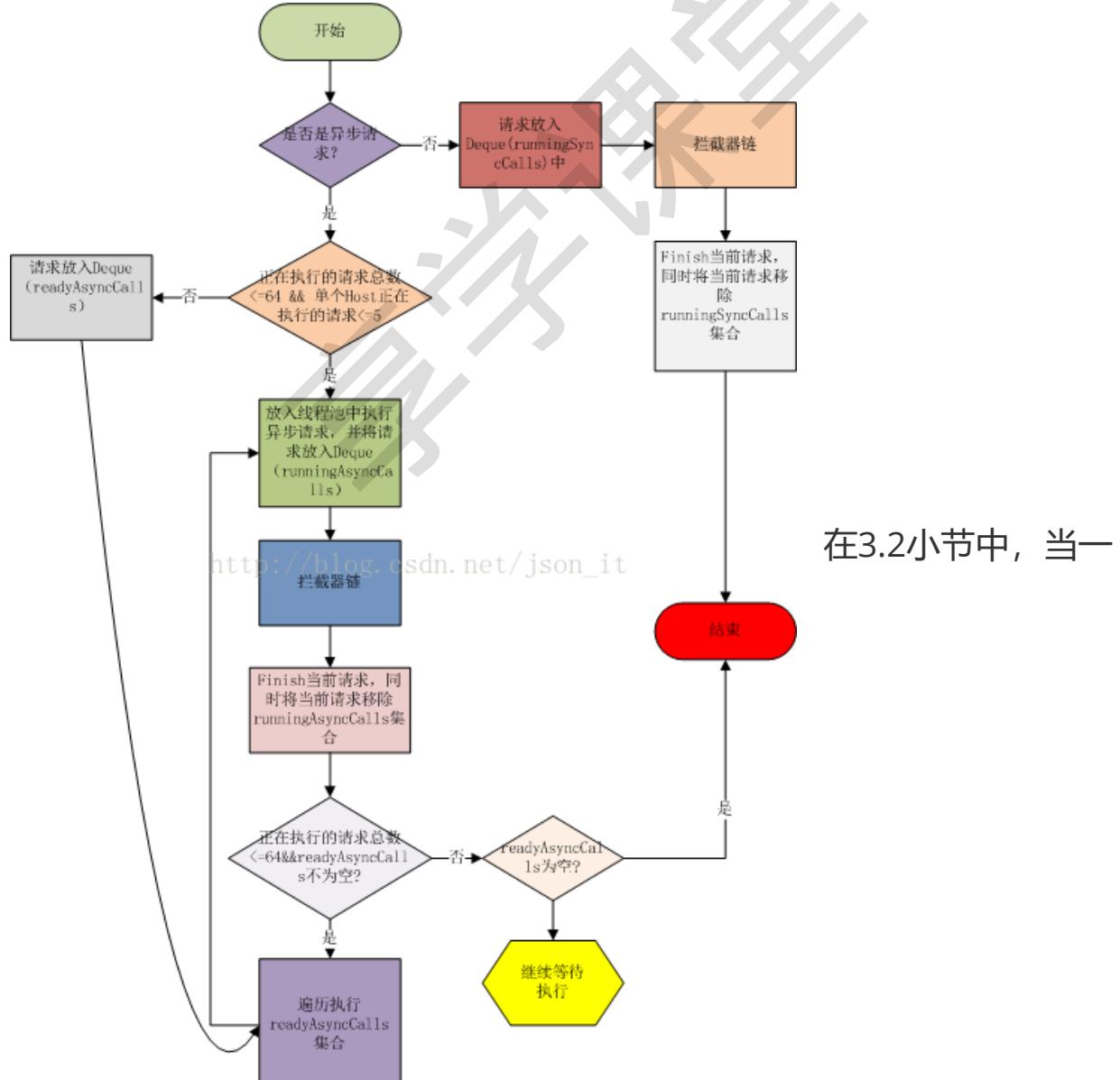
接着上面继续分析，如果可以执行，则对当前请求添加监听器等操作，然后将请求Call对象放入调度器Dispatcher中。最后由拦截器链中的各个拦截器来对该请求进行处理，返回最终的Response。

3.3、Dispatcher(调度器)

Dispatcher是保存同步和异步Call的地方，并负责执行异步AsyncCall。



如上图，针对同步请求，Dispatcher使用了一个Deque保存了同步任务；针对异步请求，Dispatcher使用了两个Deque，一个保存准备执行的请求，一个保存正在执行的请求，为什么要用两个呢？因为Dispatcher默认支持最大的并发请求是64个，单个Host最多执行5个并发请求，如果超过，则Call会先被放入到readyAsyncCall中，当出现空闲的线程时，再将readyAsyncCall中的线程移入到runningAsyncCalls中，执行请求。先看Dispatcher的流程，跟着流程读源码：



个请求是同步请求的请求的，可以看到执行了这句代码：

client.dispatcher().executed(this);根据Dispatcher源码，看一下到底发生了什么？

```
synchronized void executed(RealCall call) {  
    runningSyncCalls.add(call); } ````
```

可以看到，只是简单的将同步任务当到了runningSyncCalls集合中。在经过拦截器的处理之后，得到了响应的Response，最终会执行finally语句块：

```
``` void finished(RealCall call) {  
 finished(runningSyncCalls, call, false); } private <T>
void finished(Deque<T> calls, T call, boolean
promoteCalls) { int runningCallsCount; Runnable
idleCallback; synchronized (this) { if
(!calls.remove(call)) throw new AssertionError("Call
wasn't in-flight!"); //将请求移除集合 if (promoteCalls)
promoteCalls(); ... } ... } ````
```

对于同步请求，只是简单的将同步请求移除runningSyncCalls集合。  
promoteCalls参数是false，因此不会执行promoteCalls方法，  
promoteCalls方法用于遍历并执行异步请求待执行集合中的请求。  
Dispatcher中，同步请求的逻辑还是比较简单的。异步请求的逻辑相对麻烦一些，但也不是很复杂。在3.2小节，第二处代码是执行异步请求的逻辑，最关键的是最后依据代码：client.dispatcher().enqueue(new AsyncCall(responseCallback));紧跟着看一下enqueue方法中到底发生了什么：

```
synchronized void enqueue(AsyncCall call) { if
(runningAsyncCalls.size() < maxRequests &&
runningCallsForHost(call) < maxRequestsPerHost) {
 runningAsyncCalls.add(call);
 executorService().execute(call); } else {
 readyAsyncCalls.add(call); } } ````
```

可以看到如果正在执行的请求总数 $\leq 64$  && 单个Host正在执行的请求 $\leq 5$ , 则将请求加入到runningAsyncCalls集合中, 紧接着就是利用线程池执行该请求, 否则就将该请求放入readyAsyncCalls集合中。上面我们已经说了, AsyncCall是Runnable的子类(间接), 因此, 在线程池中最终会调用AsyncCall的execute()方法执行异步请求:

```
``` @Override protected void execute() {     boolean  
signalledCallback = false;     try {         Response  
response = getResponseWithInterceptorChain(); //拦截器链  
if (retryAndFollowUpInterceptor.isCanceled()) { //重试失败, 回调onFailure方法             signalledCallback = true;  
             responseCallback.onFailure(RealCall.this, new  
IOException("Canceled"));         } else {  
             signalledCallback = true;  
             responseCallback.onResponse(RealCall.this, response);  
         }     } catch (IOException e) {         if  
(signalledCallback) {             // Do not signal the  
callback twice!             Platform.get().log(INFO,  
"Callback failure for " + toLoggableString(), e);         } else {             eventListener.callFailed(RealCall.this,  
e);             responseCallback.onFailure(RealCall.this,  
e);         }     } finally {  
client.dispatcher().finished(this); //结束     } } ````
```

此处的执行逻辑和同步的执行逻辑基本相同, 区别在最后一句代码:
client.dispatcher().finished(this);因为这是一个异步任务, 所以会调用另外一个finish方法:

```
``` `void finished(AsyncCall call) {  
finished(runningAsyncCalls, call, true); }` private
<T> void finished(Deque<T> calls, T call, boolean
promoteCalls) { int runningCallsCount; Runnable
idleCallback; synchronized (this) { if
(!calls.remove(call)) throw new AssertionError("Call
wasn't in-flight!"); //将请求移除集合 if (promoteCalls)
promoteCalls(); ... } ... } ````
```

可以看到最后一个参数是true, 这意味着需要执行promoteCalls方法:

```
private void promoteCalls() { if
(runningAsyncCalls.size() >= maxRequests) return; //
Already running max capacity. if
(readyAsyncCalls.isEmpty()) return; // No ready calls to
promote. for (Iterator<AsyncCall> i =
readyAsyncCalls.iterator(); i.hasNext();) {
AsyncCall call = i.next(); if
(runningCallsForHost(call) < maxRequestsPerHost) {
i.remove(); runningAsyncCalls.add(call);
executorService().execute(call); } if
(runningAsyncCalls.size() >= maxRequests) return; //
Reached max capacity. } }
```

### 3.4、拦截器链

在依次介绍各个拦截器之前，先介绍一个比较重要的类：  
RealInterceptorChain，直译就是拦截器链类；这个类在什么地方会用到呢？还是3.2节，RealCall的execute方法有这么一段代码：

```
Response result = getResponseWithInterceptorChain();
```

没错，在getResponseWithInterceptorChain();方法中我们就用到了这个RealInterceptorChain类。

```
``` Response getResponsewithInterceptorChain() throws  
IOException { // Build a full stack of interceptors.  
List<Interceptor> interceptors = new ArrayList<>();  
interceptors.addAll(client.interceptors());  
interceptors.add(retryAndFollowUpInterceptor);  
interceptors.add(new  
BridgeInterceptor(client.cookieJar()));  
interceptors.add(new  
CacheInterceptor(client.internalCache()));  
interceptors.add(new ConnectInterceptor(client)); if  
(!forWebSocket) {  
interceptors.addAll(client.networkInterceptors()); }  
interceptors.add(new  
CallServerInterceptor(forWebSocket));  
Interceptor.Chain chain = new  
RealInterceptorChain(interceptors, null, null, null, 0,  
originalRequest, this, eventListener,  
client.connectTimeoutMillis(),  
client.readTimeoutMillis(), client.writeTimeoutMillis());  
return chain.proceed(originalRequest); } ```
```

可以看到，在该方法中，我们依次添加了用户自定义的interceptor、retryAndFollowUpInterceptor、BridgeInterceptor、CacheInterceptor、ConnectInterceptor、networkInterceptors、CallServerInterceptor，并将这些拦截器传递给了这个RealInterceptorChain。拦截器之所以可以依次调用，并最终再从后先前返回Response，都依赖于RealInterceptorChain的proceed方法。

```
``` public Response proceed(Request request,  
StreamAllocation streamAllocation, HttpCodec httpCodec,
RealConnection connection) throws IOException { if
(index >= interceptors.size()) throw new
AssertionError(); // call the next
interceptor in the chain. RealInterceptorChain next =
new RealInterceptorChain(interceptors, streamAllocation,
httpCodec, connection, index + 1, request, call,
eventListener, connectTimeout, readTimeout,
writeTimeout); Interceptor interceptor =
interceptors.get(index); Response response =
interceptor.intercept(next); return
response; } ```
```

该方法最核心的代码就是中间的这几句，执行当前拦截器的Intercept方法，并调用下一个 (index+1) 拦截器。下一个 (index+1) 拦截器的调用依赖于当前拦截器的Intercept方法中，对RealInterceptorChain的proceed方法的调用：

```
``` response = realChain.proceed(request,  
streamAllocation, null, null); ```
```

可以看到当前拦截器的Response依赖于下一个拦截器的Intercept的Response。因此，就会沿着这条拦截器链依次调用每一个拦截器，当执行到最后一个拦截器之后，就会沿着相反的方向依次返回Response，最终得到我们需要的“终极版”Response。

3.4.1、重试及 followup拦截器

想
識
思
學

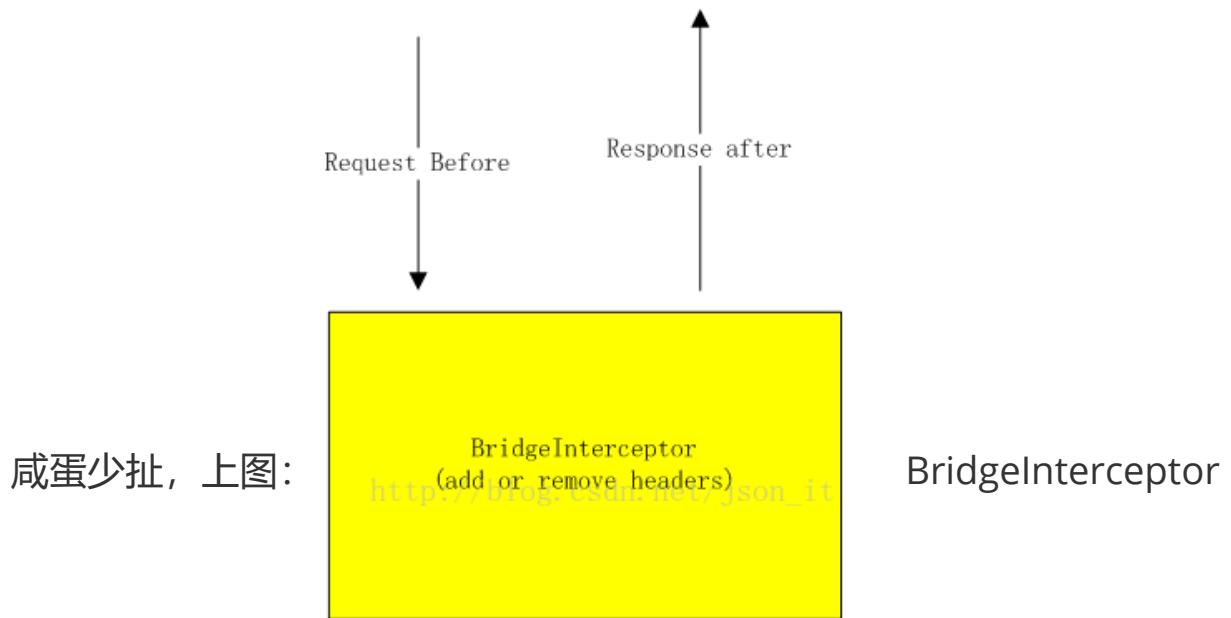
```
``` @Override public Response intercept(Chain chain) throws IOException { Request request = chain.request(); //获取Request对象 RealInterceptorChain realChain = (RealInterceptorChain) chain; //获取拦截器链对象，用于后面的chain.proceed(...)方法 call call = realChain.call(); EventListener eventListener = realChain.eventListener(); //监听器 streamAllocation = new StreamAllocation(client.connectionPool(), createAddress(request.url()), call, eventListener, callStackTrace); int followUpCount = 0; Response priorResponse = null; while (true) { //循环 if (canceled) { streamAllocation.release(); throw new IOException("Canceled"); } Response response; boolean releaseConnection = true; try { response = realChain.proceed(request, streamAllocation, null, null); //调用下一个拦截器 releaseConnection = false; } catch (RouteException e) { // The attempt to connect via a route failed. The request will not have been sent. if (!recover(e.getLastConnectException(), false, request)) { //路由异常，尝试恢复，如果再失败就抛出异常 throw e.getLastConnectException(); } releaseConnection = false; continue; //继续重试 } catch (IOException e) { // An attempt to communicate with a server failed. The request may have been sent. boolean requestSendStarted = !(e instanceof ConnectionShutdownException); if (!recover(e, requestSendStarted, request)) { //连接关闭异常，尝试恢复 throw e; } releaseConnection = false; continue; //继续重试 } finally { //
```

```
we're throwing an unchecked exception. Release any
resources. if (releaseConnection) {
 streamAllocation.streamFailed(null);
 streamAllocation.release();
}
}
// Attach
the prior response if it exists. Such responses never
have a body. if (priorResponse != null)
{ //前一个重试得到的Response
 response = response.newBuilder()
 .priorResponse(priorResponse.newBuilder()
 .body(null)
 .build())
 .build();
}
//Figures out the HTTP
request to make in response to receiving {@code
userResponse}. This will //either add
authentication headers, follow redirects or handle a
client request timeout. If a //follow-up
is either unnecessary or not applicable, this returns
null. // followUpRequest方法的主要作用就是为新的
 // followUpRequest添加验证头等内容
 Request followUp
= followUpRequest(response);
if (followUp == null) { //如果一个请求得到的响应code是200,
则followUp是为null的。
if
(!forwebsocket) { streamAllocation.release(); } return
response; }
closeQuietly(response.body()); //-----
-----异常处理-----
----- // if (++followUpCount >
MAX_FOLLOW_UPS) { //超过最大的次数,抛出异常
streamAllocation.release(); throw new
ProtocolException("Too many follow-up requests: " +
followUpCount); } if (followUp.body()
instanceof UnrepeatableRequestBody) {
streamAllocation.release(); } throw new
HttpRetryException("Cannot retry streamed HTTP body",
response.code()); } if (!sameConnection(response,
followUp.url())) { streamAllocation.release();
streamAllocation = new
```

```
streamAllocation(client.connectionPool(),
createAddress(followUp.url()), call, eventListener,
callStackTrace); } else if
(streamAllocation.codec() != null) { throw new
IllegalStateException("Closing the body of " + response +
" didn't close its backing stream. Bad interceptor?");
} //-----
----- request =
followUp;//得到处理之后的Request，以用来继续请求，在哪继续请求？
肯定还是沿着拦截器链继续搞呗 priorResponse =
response;//由priorResponse持有 } } } ``
```

该拦截器主要的作用就是重试及followup(这个followup咋翻译比较贴切呢？)。当一个请求由于各种原因失败了，如果是路由或者连接异常，则尝试恢复，否则，根据响应码（ResponseCode），followup方法会对Request进行再处理以得到新的Request，然后沿着拦截器链继续新的Request。当然，如果responseCode是200的话，这些过程就结束了。注意看注释。

### 3.4.2、BridgeInterceptor



咸蛋少扯，上图：

`BridgeInterceptor`

的主要作用就是为请求 (request before) 添加请求头，为响应 (Response Before) 添加响应头。看源码：

想  
識  
思  
學

```
``` @Override public Response intercept(Chain chain)
throws IOException { Request userRequest =
chain.request(); Request.Builder requestBuilder =
userRequest.newBuilder(); //-----
request-----  

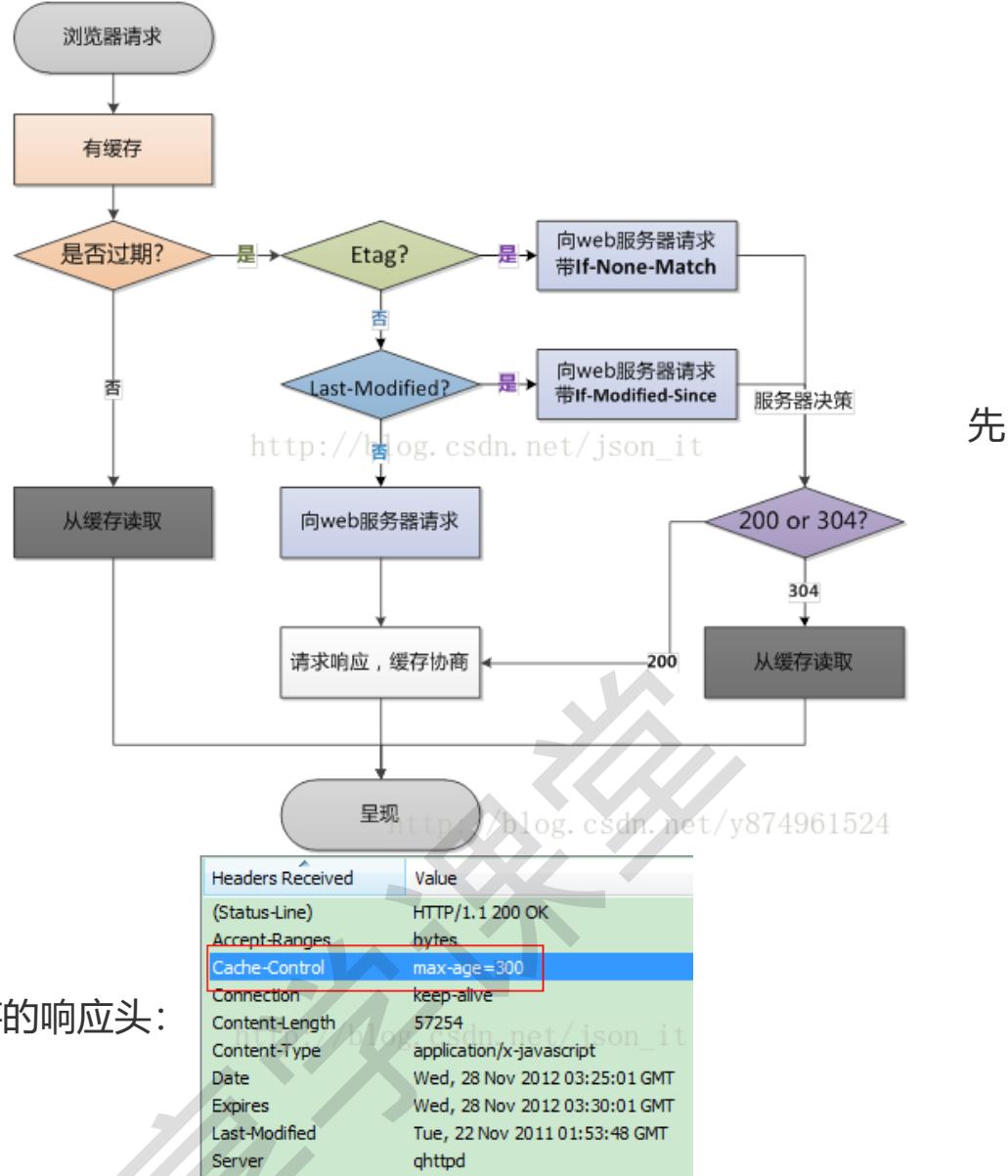
RequestBody body = userRequest.body(); if (body != null) { MediaType contentType = body.contentType();
if (contentType != null) {//添加Content-Type请求头
requestBuilder.header("Content-Type",
contentType.toString()); } long contentLength
= body.contentLength(); if (contentLength != -1) {
requestBuilder.header("Content-Length",
Long.toString(contentLength));
requestBuilder.removeHeader("Transfer-Encoding"); }
else { requestBuilder.header("Transfer-Encoding",
"chunked");//分块传输
requestBuilder.removeHeader("Content-Length"); }
} if (userRequest.header("Host") == null) {
requestBuilder.header("Host",
hostHeader(userRequest.url(), false)); } if
(userRequest.header("Connection") == null) {
requestBuilder.header("Connection", "Keep-Alive");
// If we add an "Accept-Encoding: gzip" header field
we're responsible for also decompressing // the
transfer stream. boolean transparentGzip = false;
if (userRequest.header("Accept-Encoding") == null &&
userRequest.header("Range") == null) {
transparentGzip = true;
requestBuilder.header("Accept-Encoding", "gzip"); }
List<Cookie> cookies =
cookieJar.loadForRequest(userRequest.url()); if
(!cookies.isEmpty()) {
requestBuilder.header("Cookie", cookieHeader(cookies));
} if (userRequest.header("User-Agent") == null) {
requestBuilder.header("User-Agent", Version.userAgent());
} Response networkResponse =
chain.proceed(requestBuilder.build()); //-----
-----response-----
```

```
-----      HttpHeaders.receiveHeaders(cookieJar,
userRequest.url(), networkResponse.headers());//保存cookie
Response.Builder responseBuilder =
networkResponse.newBuilder()
.request(userRequest);      if (transparentGzip        &&
"gzip".equalsIgnoreCase(networkResponse.header("Content-
Encoding"))           &&
HttpHeaders.hasBody(networkResponse)) {          GzipSource
responseBody = new
GzipSource(networkResponse.body().source());      Headers
strippedHeaders = networkResponse.headers().newBuilder()
.removeAll("Content-Encoding")//Content-
Encoding、Content-Length不能用于Gzip解压缩
.removeAll("Content-Length")           .build();
responseBuilder.headers(strippedHeaders);      string
contentType = networkResponse.header("Content-Type");
responseBuilder.body(new RealResponseBody(contentType,
-1L, okio.buffer(responseBody))); }      return
responseBuilder.build(); }
```

这个拦截器的源码还是很简单的，不再详细叙述。

3.4.3、CacheInterceptor

在解析CacheInterceptor之前，先看一张关于Http缓存机制的图片（来源于网络）：



看一下缓存的响应头：

(本模块前两个图均来自于<http://blog.csdn.net/y874961524/article/details/61419716>, 感谢)

几个相关的字段先解释一下（估计都知道）：

Cache-control：标明缓存的最大存活时常；

Date：服务器告诉客户端，该资源的发送时间；

Expires：表示过期时间（该字段是1.0的东西，当cache-control和该字段同时存在的条件下，cache-control的优先级更高）；

Last-Modified：服务器告诉客户端，资源的最后修改时间；还有一个字段，这个图没给出，就是E-Tag：当前资源在服务器的唯一标识，可用于判断资源的内容是否被修改了。

除以上响应头字段以外，还需了解两个相关的Request请求头：If-Modified-since、If-none-Match。这两个字段是和Last-Modified、E-Tag配合使用的。大致流程如下：

服务器收到请求时，会在200 OK中回送该资源的Last-Modified和ETag头（服务器支持缓存的情况下才会有这两个头哦），客户端将该资源保存在cache中，并记录这两个属性。当客户端需要发送相同的请求时，根据Date + Cache-control来判断是否缓存过期，如果过期了，会在请求中携带If-Modified-Since和If-None-Match两个头。两个头的值分别是响应中Last-Modified和ETag头的值。服务器通过这两个头判断本地资源未发生变化，客户端不需要重新下载，返回304响应。看源码之前，先看几个与CacheInterceptor相关的比较重要的几个类：



CacheStrategy是一个缓存策略类，该类告诉CacheInterceptor是使用缓存还是使用网络请求； Cache是封装了实际的缓存操作； DiskLruCache:Cache基于DiskLruCache； 下面看一下CacheInterceptor的源码：

想
識
思
學

```
``` @Override public Response intercept(Chain chain)
throws IOException { Response cacheCandidate = cache
!= null ? cache.get(chain.request())//以request的
url而来key, 获取缓存 : null; long now =
System.currentTimeMillis(); //缓存策略类, 该类决定了是使用
缓存还是进行网络请求 CacheStrategy strategy = new
CacheStrategy.Factory(now, chain.request(),
cacheCandidate).get(); Request networkRequest =
strategy.networkRequest;//网络请求, 如果为null就代表不用进行网
络请求 Response cacheResponse =
strategy.cacheResponse;//缓存响应, 如果为null, 则代表不使用缓存
 if (cache != null) {//根据缓存策略, 更新统计指标: 请求次数、
使用网络请求次数、使用缓存次数
 cache.trackResponse(strategy); } //缓存不可用, 关闭
 if (cacheCandidate != null && cacheResponse == null) {
 closeQuietly(cacheCandidate.body()); // The cache
candidate wasn't applicable. Close it. } //如果既无网
络请求可用, 又没有缓存, 则返回504错误 // If we're forbidden
from using the network and the cache is insufficient,
fail. if (networkRequest == null && cacheResponse ==
null) { return new Response.Builder()
.request(chain.request())
.protocol(Protocol.HTTP_1_1) .code(504)
.message("Unsatisfiable Request (only-if-cached)")
.body(Util.EMPTY_RESPONSE)
.sentRequestAtMillis(-1L)
.receivedResponseAtMillis(System.currentTimeMillis())
.build(); } // If we don't need the network,
we're done.缓存可用, 直接返回缓存 if (networkRequest ==
null) { return cacheResponse.newBuilder()
.cacheResponse(stripBody(cacheResponse))
.build(); } Response networkResponse = null;
try { networkResponse =
chain.proceed(networkRequest);//进行网络请求, 得到网络响应
} finally { // If we're crashing on I/O or
otherwise, don't leak the cache body. if
(networkResponse == null && cacheCandidate != null) {
 closeQuietly(cacheCandidate.body()); } }
```

```
//HTTP_NOT_MODIFIED缓存有效，合并网络请求和缓存 // If we
have a cache response too, then we're doing a conditional
get. if (cacheResponse != null) { if
(networkResponse.code() == HTTP_NOT_MODIFIED) {
Response response = cacheResponse.newBuilder()
.headers(combine(cacheResponse.headers(),
networkResponse.headers()))
.sentRequestAtMillis(networkResponse.sentRequestAtMillis(
)))
.receivedResponseAtMillis(networkResponse.receivedRespon
eAtMillis())
.cacheResponse(stripBody(cacheResponse))
.networkResponse(stripBody(networkResponse))
.build(); networkResponse.body().close();
// Update the cache after combining headers but before
stripping the // Content-Encoding header (as
performed by initContentStream()).
cache.trackConditionalCacheHit();
cache.update(cacheResponse, response); //更新缓存
return response; } else {
closeQuietly(cacheResponse.body()); }
Response response = networkResponse.newBuilder()
.cacheResponse(stripBody(cacheResponse))
.networkResponse(stripBody(networkResponse))
.build(); if (cache != null) { //有响应体 & 可缓存
if (HttpHeaders.hasBody(response) &&
CacheStrategy.isCacheable(response, networkRequest)) {
// Offer this request to the cache.
CacheRequest cacheRequest = cache.put(response);
return cachewritingResponse(cacheRequest, response); //写缓
存 } if
(HttpMethod.invalidateCache(networkRequest.method()))
{//判断缓存的有效性 try {
cache.remove(networkRequest); } catch (IOException
ignored) { // The cache cannot be written.
} } return response; } ``
```

上面源码中的注释已经解释的很清楚了，下面再简单的说一下流程：

根据缓存策略类返回的结果：

- 1、如果网络不可用并且无可用的有效缓存，则返回504错误；
- 2、继续，如果不需要网络请求，则直接使用缓存；
- 3、继续，如果需要网络可用，则进行网络请求；
- 4、继续，如果有缓存，并且网络请求返回HTTP\_NOT\_MODIFIED，说明缓存还是有效的，则合并网络响应和缓存结果。同时更新缓存；
- 5、继续，如果没有缓存，则写入新的缓存；我们可以看到，CacheStrategy在CacheInterceptor中起到了很关键的作用。该类决定了是网络请求还是使用缓存。该类最关键的代码是getCandidate（）方法：

想  
識  
思  
學

```
``` private CacheStrategy getCandidate() { // No
cached response. if (cacheResponse == null) {//没有缓
存, 直接网络请求 return new CacheStrategy(request,
null); } // Drop the cached response if it's
missing a required handshake. if (request.isHttps()
&& cacheResponse.handshake() == null) {//https, 但没有握手,
直接网络请求 return new CacheStrategy(request,
null); } // If this response shouldn't have
been stored, it should never be used // as a
response source. This check should be redundant as long
as the // persistence store is well-behaved and the
rules are constant. if (!isCacheable(cacheResponse,
request)) {//不可缓存, 直接网络请求 return new
CacheStrategy(request, null); } CacheControl
requestCaching = request.cacheControl(); if
(requestCaching.noCache() || hasConditions(request)) {
//请求头nocache或者请求头包含If-Modified-Since或者If-
None-Match //请求头包含If-Modified-Since或者If-None-
Match意味着本地缓存过期, 需要服务器验证 //本地缓存是不是还
能继续使用 return new CacheStrategy(request, null);
} CacheControl responseCaching =
cacheResponse.cacheControl(); if
(responseCaching.immutable()) {//强制使用缓存 return
new CacheStrategy(null, cacheResponse); } long
ageMillis = cacheResponseAge(); long freshMillis =
computeFreshnessLifetime(); if
(requestCaching.maxAgeSeconds() != -1) {
freshMillis = Math.min(freshMillis,
SECONDS.toMillis(requestCaching.maxAgeSeconds())); }
long minFreshMillis = 0; if
(requestCaching.minFreshSeconds() != -1) {
minFreshMillis =
SECONDS.toMillis(requestCaching.minFreshSeconds()); }
long maxStaleMillis = 0; if
(!responseCaching.mustRevalidate() &&
requestCaching.maxStaleSeconds() != -1) {
maxStaleMillis =
SECONDS.toMillis(requestCaching.maxStaleSeconds()); }
```

```
        //可缓存，并且ageMillis + minFreshMillis <
freshMillis + maxStaleMillis      // (意味着虽过期，但可用,
只是会在响应头添加warning)      if
(!responseCaching.noCache() && ageMillis + minFreshMillis
< freshMillis + maxStaleMillis) {      Response.Builder
builder = cacheResponse.newBuilder();      if
(ageMillis + minFreshMillis >= freshMillis) {
builder.addHeader("Warning", "110 HttpURLConnection
\"Response is stale\"");      }      long
oneDayMillis = 24 * 60 * 60 * 1000L;      if (ageMillis
> oneDayMillis && isFreshnessLifetimeHeuristic()) {
builder.addHeader("Warning", "113 HttpURLConnection
\"Heuristic expiration\"");      }      return new
CacheStrategy(null, builder.build());//使用缓存      }
// Find a condition to add to the request. If the
condition is satisfied, the response body      // will
not be transmitted.      String conditionName;
String conditionValue;      //流程走到这，说明缓存已经过期了
//添加请求头: If-Modified-Since或者If-None-Match
//etag与If-None-Match配合使用      //lastModified与If-
Modified-Since配合使用      //前者和后者的值是相同的      //区别在于前者是响应头，后者是请求头。      //后者用于服务器进行资源比
对，看看是资源是否改变了。      // 如果没有，则本地的资源虽过期还是
可以用的      if (etag != null) {      conditionName =
"If-None-Match";      conditionValue = etag;      }
else if (lastModified != null) {      conditionName =
"If-Modified-Since";      conditionValue =
lastModifiedString;      } else if (servedDate != null) {
conditionName = "If-Modified-Since";
conditionValue = servedDateString;      } else {
return new CacheStrategy(request, null); // No condition!
Make a regular request.      }      Headers.Builder
conditionalRequestHeaders =
request.headers().newBuilder();
Internal.instance.addLenient(conditionalRequestHeaders,
conditionName, conditionValue);      Request
conditionalRequest = request.newBuilder()
.headers(conditionalRequestHeaders.build())

```

```
.build();      return new  
CacheStrategy(conditionalRequest, cacheResponse);    }  
```
```

大致流程如下： (if-else的关系呀)

- 1、没有缓存，直接网络请求；
- 2、如果是https，但没有握手，直接网络请求；
- 3、不可缓存，直接网络请求；
- 4、请求头nocache或者请求头包含If-Modified-Since或者If-None-Match，则需要服务器验证本地缓存是不是还能继续使用，直接网络请求；
- 5、可缓存，并且ageMillis + minFreshMillis < freshMillis + maxStaleMillis（意味着虽过期，但可用，只是会在响应头添加warning），则使用缓存；
- 6、缓存已经过期，添加请求头：If-Modified-Since或者If-None-Match，进行网络请求；

#### 3.4.4、ConnectInterceptor (核心，连接池)

ConnectInterceptor器如其名，是一个连接相关的拦截器。这个拦截器是这几个拦截器里面代码最少的。但是少并不意味着很简单。先看一下ConnectIntercepotor中比较重要的几个类及其含义：



RouteDataBase：这是一个关于路由信息的白名单和黑名单类，处于黑名单的路由信息会被避免不必要的尝试；

RealConnecton：Connect子类，主要实现连接的建立等工作；  
ConnectionPool:连接池，实现连接的复用；

这里再说一下Connection和Stream的关系：Http1.x是1:1的关系，而Http2是1对多的关系。就是说一个http1.x连接只能被一个请求使用，而一个Http2连接是对应多个Stream的，多个Stream的意思是Http2连接支持并发请求，即一个连接可以被多个请求同时使用的。还有，Http1.1的keep-alive机制的作用是保证连接使用完不关闭，当下一次请求与连接的Host相同的时候，连接可以直接使用，不用再次创建（节省资源，提高了性能）。

StreamAllocation：直译就是流分配。流是什么呢？我们知道Connection是一个连接远程服务器的物理Socket连接，而Stream则是基于Connection的逻辑Http 请求/响应对。StreamAllocation会通过ConnectPool获取或者新生成一个RealConnection来得到一个连接到Server的Connection连接，同时会生成一个HttpCodec用于下一个CallServerInterceptor，以完成最终的请求；HttpCodec：Encodes HTTP requests and decodes HTTP responses。（源码注释哦）。针对不同的版本，OkHttp为我们提供了HttpCodec1（Http1.x）和HttpCodec2(Http2)。一句话概括就是：分配一个Connection和HttpCodec，为最终的请求做准备。

```
``` /** Opens a connection to the target server and
proceeds to the next interceptor. */ public final class
ConnectInterceptor implements Interceptor { public final
OkHttpClient client; public
ConnectInterceptor(OkHttpClient client) { this.client
= client; } @Override public Response intercept(Chain
chain) throws IOException { RealInterceptorChain
realChain = (RealInterceptorChain) chain; Request
request = realChain.request(); StreamAllocation
streamAllocation = realChain.streamAllocation(); // we need the network to satisfy this request. Possibly for
validating a conditional GET. // 我们需要网络来满足这个请
求。可能是为了验证一个条件GET请求（缓存验证等）。 boolean
doExtensiveHealthChecks =
!request.method().equals("GET"); HttpCodec httpCodec =
streamAllocation.newStream(client, chain,
doExtensiveHealthChecks); RealConnection connection =
streamAllocation.connection(); return
realChain.proceed(request, streamAllocation, httpCodec,
connection); } } ```
```

代码量是不是很少？是的。表面上看起来很少，实际上大部分的功能都被封装到其他的类里面去了，此处只是调用。所以为了代码的可读性和可维护性，该封装的还是乖乖的封装吧。核心代码就两行：

```
``` HttpCodec httpCodec =
streamAllocation.newStream(client, chain,
doExtensiveHealthChecks); RealConnection connection =
streamAllocation.connection(); ```
```

可以看出，主要的工作是由StreamAllocation完成。我们来看看这个StreamAllocation的newStream和connection（）到底做了什么。

```
``` public HttpCodec newStream(          OkHttpClient
client, Interceptor.Chain chain, boolean
doExtensiveHealthChecks) {      int connectTimeout =
chain.connectTimeoutMillis();      int readTimeout =
chain.readTimeoutMillis();      int writeTimeout =
chain.writeTimeoutMillis();      boolean
connectionRetryEnabled =
client.retryOnConnectionFailure();      try {
RealConnection resultConnection =
findHealthyConnection(connectTimeout, readTimeout,
writeTimeout, connectionRetryEnabled,
doExtensiveHealthChecks);      HttpCodec resultCodec =
resultConnection.newCodec(client, chain, this);
synchronized (connectionPool) {      codec =
resultCodec;      return resultCodec;      }      } catch
(IOException e) {      throw new RouteException(e);      }
} ````
```

可以看到，最关键的一步就是`findHealthyConnection`，这个方法的主要的作用就是找到一个可用的连接（如果连接不可用，这个过程会一直持续哦）。

```

``` private RealConnection findHealthyConnection(int
connectTimeout, int readTimeout, int writeTimeout,
boolean connectionRetryEnabled, boolean
doExtensiveHealthChecks) throws IOException {
while (true) { RealConnection candidate =
findConnection(connectTimeout, readTimeout, writeTimeout,
connectionRetryEnabled); // If this is a
brand new connection, we can skip the extensive health
checks.如果是一个新的连接, 直接返回就好了 synchronized
(connectionPool) { if (candidate.successCount ==
0) { return candidate; } } //
Do a (potentially slow) check to confirm that the pooled
connection is still good. If it // isn't, take it
out of the pool and start again. if
(!candidate.isHealthy(doExtensiveHealthchecks)) { //判断连接
是否好使 noNewStreams(); //连接不好使的话, 移除连接池
continue; //不healthy, 就一直持续的呀 }
return
candidate; } } ```

```

上述代码还是很容易理解的, 唯一让我有点费解的就是这个noNewStream方法。刚开始看名字有点蒙圈。啥叫noNewStream嘞, 看源码 (其实应该先看findConnection的源码的, 但是先搞懂这个地方, 对后面的理解有益无害的) :

```

``` public void noNewStreams() {      Socket socket;
Connection releasedConnection;      synchronized
(connectionPool) {      releasedConnection = connection;
socket = deallocate(true, false, false); // noNewStreams,
released, streamFinished核心方法      if
(connection != null) releasedConnection = null;      }
closeQuietly(socket); //关闭socket      if
(releasedConnection != null) {
eventListener.connectionReleased(call,
releasedConnection); //监听回调      }  } ```

```

上述关键代码是deallocate:

```

``` private Socket deallocate(boolean noNewStreams,
boolean released, boolean streamFinished) { assert
(Thread.holdsLock(connectionPool)); //以noNewStreams为
true, released为false, streamFinished为false;为例 if
(streamFinished) { this.codec = null; } if
(released) { this.released = true; } Socket
socket = null; if (connection != null) { if
(noNewStreams) { //noNewStreams是RealConnection的属
性, 源码的注释是这么说的: //如果为true, 则这个连接就不会再
创建新的Stream了, 一旦设置成true, 就会一直是true //搜索整个
源码, 该属性设置的地方如下: //evitAll: 关闭和移除连接池
中所有的空闲连接 (如果连接空闲(即连接上的Stream数为0), 则
noNewStreams为true); //pruneAndGetAllocationCount: 移除内存泄漏的连接及获取连接的
Stream分配数; //streamFailed: Stream分配失败;
//综上, 这个属性的作用是禁止无效连接创建新的Stream的
connection.noNewStreams = true; } if
(this.codec == null && (this.released ||
connection.noNewStreams)) { release(connection); //
释放Connection承载的StreamAllocations资源
(connection.allocations) if
(connection.allocations.isEmpty()) {
connection.idleAtNanos = System.nanoTime();
//connectionBecameIdle: 通知线程池该连接是空闲连接, 可以移除或者
作为待移除对象。 if
(Internal.instance.connectionBecameIdle(connectionPool,
connection)) { socket = connection.socket();
} } connection = null; }
return socket; //返回待关闭的Socket对象 } ```

```

需要说的都写在了上面源码的注释里面了, 不再多说了。接着看findConnection方法, 好吧, 继续, 源码有点长, 不过我都给注释了, 看起来应该也不会很难。

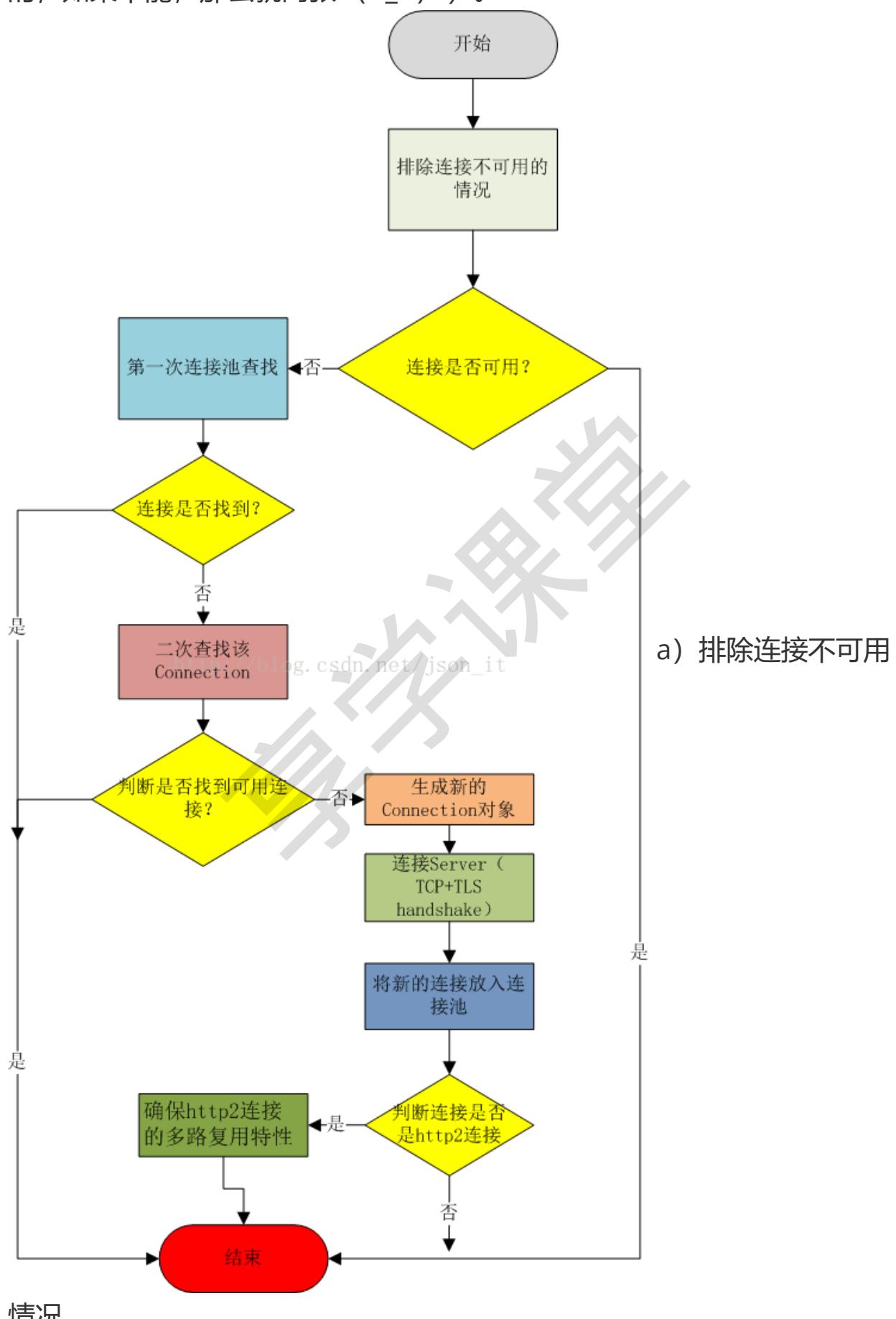
想  
識  
思  
學

```
``` private RealConnection findConnection(int
connectTimeout, int readTimeout, int writeTimeout,
boolean connectionRetryEnabled) throws IOException {
boolean foundPooledConnection = false;      RealConnection
result = null;      Route selectedRoute = null;
Connection releasedConnection;      Socket toClose;
synchronized (connectionPool) {      //-----排除异常情
况-----      if (released) throw new
IllegalStateException("released");      if (codec !=
null) throw new IllegalStateException("codec != null");
      if (canceled) throw new IOException("Canceled");
// Attempt to use an already-allocated connection. We
need to be careful here because our      // already-
allocated connection may have been restricted from
creating new streams.      releasedConnection =
this.connection;      //这个方法的作用，与deallocate作用一样
      //如果连接不能创建stream，则释放资源，返回待关闭的close
Socket      toClose = releaseIfNoNewStreams();      //经过
releaseIfNoNewStreams，如果connection不为null，则连接是可用的
      if (this.connection != null) {      // we had an
already-allocated connection and it's good.      //存在
可使用的已分配连接      result = this.connection;
releasedConnection = null;//为null值，则说明这个连接是有效的
}      if (!reportedAcquired) {      // If the
connection was never reported acquired, don't report it
as released!      releasedConnection = null;  }
      if (result == null) { //没有可使用的连接，去连接池中找
// Attempt to get a connection from the pool. //首先通过
ConnectionPool,Address,StreamAllocation从连接池获取连接，
      // 连接池后面会单独讲解*****
Internal.instance.get(connectionPool, address, this,
null); //ConnectionPool,Address,StreamAllocation,Route
      if (connection != null) {
foundPooledConnection = true;      result =
connection;      } else {      selectedRoute =
route;      }  }  }  closeQuietly(toClose);
if (releasedConnection != null) {
eventListener.connectionReleased(call,
```

```
releasedConnection); } if (foundPooledConnection) {  
    eventListener.connectionAcquired(call, result); }  
    if (result != null) { // If we found an already-  
allocated or pooled connection, we're done. return  
result; //找到了一个已分配或者连接池中的连接, 此过程结束, 返回 }  
    //否则, 我们需要一个路由信息, 这是一个阻塞的操作 // If we  
need a route selection, make one. This is a blocking  
operation. boolean newRouteSelection = false; if  
(selectedRoute == null && (routeSelection == null ||  
!routeSelection.hasNext())) { newRouteSelection =  
true; routeSelection = routeSelector.next(); }  
    synchronized (connectionPool) { if (canceled)  
throw new IOException("Canceled"); if  
(newRouteSelection) { // Now that we have a set of  
IP addresses, make another attempt at getting a  
connection from // the pool. This could match due  
to connection coalescing. //提供更加全面的路由信息, 再  
次从连接池中获取连接 List<Route> routes =  
routeSelection.getAll(); for (int i = 0, size =  
routes.size(); i < size; i++) { Route route =  
routes.get(i);  
Internal.instance.get(connectionPool, address, this,  
route); if (connection != null) {  
foundPooledConnection = true; result =  
connection; this.route = route;  
break; } } /*实在是没找到, 只能  
生成新的连接*****了 if (!foundPooledConnection) {  
    if (selectedRoute == null) { selectedRoute =  
routeSelection.next(); } // Create a  
connection and assign it to this allocation immediately.  
This makes it possible // for an asynchronous  
cancel() to interrupt the handshake we're about to do.  
    route = selectedRoute; refusedStreamCount =  
0; result = new RealConnection(connectionPool,  
selectedRoute); acquire(result, false); //添加  
connection的StreamAllocation添加到connection.allocations集  
合中***** } } // If we found a pooled  
connection on the 2nd time around, we're done. //如果连
```

```
接是从连接池中找到的，说明是可复用的。不是新生成的，因为新生成的连接，      // 需要去连接服务器之后才能可用呀      if
(foundPooledConnection) {
eventListener.connectionAcquired(call, result);
return result; }      // Do TCP + TLS handshakes. This
is a blocking operation.//连接Server      result.connect(
    connectTimeout, readTimeout, writeTimeout,
connectionRetryEnabled, call, eventListener);
routeDatabase().connected(result.route()); //将路由信息添加到
routeDatabase中。      Socket socket = null;
synchronized (connectionPool) {      reportedAcquired =
true;      // Pool the connection.
Internal.instance.put(connectionPool, result); //将新生成的
连接放入连接池中      // If another multiplexed connection
to the same address was created concurrently, then
// release this connection and acquire that one.      //如
果是一个http2连接，由于http2连接应具有多路复用特性，      // 因
此，我们需要确保http2连接的多路复用特性      if
(result.isMultiplexed()) {      //deduplicate:确保http2
连接的多路复用特性，重复的连接将被剔除      socket =
Internal.instance.deduplicate(connectionPool, address,
this);      result = connection;      }      }
closeQuietly(socket);
eventListener.connectionAcquired(call, result);      return
result; } ``
```

上述代码加了很多注释，可以看一下。为了更加快速的了解其过程，画了一个流程图，跟着流程图来一步一步的解析（没有什么是一张图解决不了的，如果不能，那么就两张O(∩_∩)O）。



```
``` private Socket releaseIfNoNewStreams() {     assert  
 (Thread.holdsLock(connectionPool)); RealConnection
 allocatedConnection = this.connection; if
 (allocatedConnection != null &&
 allocatedConnection.noNewStreams) { return
 deallocate(false, false, true); } return null; }
```
```

这个方法是说如果连接处于noNewStream状态，则释放该连接。否则，该连接是可用的。关于noNewStream和deallocate方法前面已经解释的很清楚了。

b) 判断连接是否可用

经过releaseIfNoNewStreams方法，如果connection不为null，则一定是可用的。

```
``` //经过releaseIfNoNewStreams, 如果connection不为null, 则连  
接是可用的 if (this.connection != null) { // we
had an already-allocated connection and it's good.
//存在可使用的已分配连接 result = this.connection;
releasedConnection = null;//为null值, 则说明这个连接是有效
的 } ```
```

### c) 第一次连接池查找（没有提供路由信息）

```
Internal.instance.get(connectionPool, address, this,
null);//ConnectionPool,Address,StreamAllocation,Route
if (connection != null) {
foundPooledConnection = true; result =
connection; }
```

如果查找到了，则将查找到的连接赋值给result。 d) 遍历路由表，进行二次查找

```
``` List<Route> routes = routeSelection.getAll();
for (int i = 0, size = routes.size(); i < size; i++) {
    Route route = routes.get(i);
Internal.instance.get(connectionPool, address, this,
route); if (connection != null) {
foundPooledConnection = true; result =
connection; this.route = route;
break; } ````
```

f) 如果还是没找到，则只能创建新的连接了

```
``` result = new RealConnection(connectionPool,
selectedRoute); acquire(result, false); //添加
connection的StreamAllocation添加到connection.allocations集
合中***** ````
```

g) 新的连接，连接服务器

```
// Do TCP + TLS handshakes. This is a blocking operation.//
连接Server（connect方法涉及Socket的建立等） result.connect(
connectTimeout, readTimeout, writeTimeout,
connectionRetryEnabled, call, eventListener);
routeDatabase().connected(result.route()); //将路由信息添加到
routeDatabase中。
```

h) 新的连接放入线程池

```
// Pool the connection.
Internal.instance.put(connectionPool, result); //将新生成的
连接放入连接池中
```

i) 如果连接是一个HTTP2连接，则需要确保多路复用的特性

```

``` //如果是一个http2连接，由于http2连接应具有多路复用特性，  

// 因此，我们需要确保http2连接的多路复用特性 if  

(result.isMultiplexed()) { //deduplicate:确保http2  

连接的多路复用特性，重复的连接将被剔除 socket =  

Internal.instance.deduplicate(connectionPool, address,  

this); result = connection; } ```

```

在Connectinterceptor中，起到关键作用的就是ConnectionPool，既然这么关键我们就来看看这个连接池吧。



----- 在目前的版本下，连接池默认是可以保持5个空闲的连接。这些空闲的连接如果超过5分钟不被使用，则将被连接池移除。当然，这些默认的数值在未来的okhttp版本中，会被改变的。另外，这两个数值支持开发人员修改。 -----

----- ConnectionPool中比较关键的几个点，线程池（ThreadPoolExecutor）、队列（Deque）、路由记录表；线程池：用于支持连接池的cleanup任务，清除idle线程；队列：存放待复用的连接；路由记录表：前面已讲，不再叙述；对于连接池，开发人员最感兴趣的肯定的：存、取、清除；

a) 存

```

void put(RealConnection connection) { assert  

(Thread.holdsLock(this)); if (!cleanupRunning) {  

cleanupRunning = true;  

executor.execute(cleanupRunnable); }  

connections.add(connection); }

```

可以看到，在放入连接到connections(Deque)之前，可能是需要执行连接池的“清洁”任务的。连接存入连接池的操作很简单，主要看一下这个cleanUp到底做了些什么？

想
識
思
學

```
``` long cleanup(long now) {     int inUseConnectionCount
= 0; int idleConnectionCount = 0; RealConnection
longestIdleConnection = null; long
longestIdleDurationNs = Long.MIN_VALUE; // Find
either a connection to evict, or the time that the next
eviction is due. synchronized (this) {
 for
 (Iterator<RealConnection> i = connections.iterator());
 i.hasNext();) { RealConnection connection =
 i.next(); // If the connection is in use, keep
 searching. if
 (pruneAndGetAllocationCount(connection, now) > 0) {
 inUseConnectionCount++; //线程池中处于使用状态的连接数
 continue; } idleConnectionCount++; //处于
 空闲状态的连接数 // If the connection is ready to be
 evicted, we're done. long idleDurationNs = now -
 connection.idleAtNanos; //寻找空闲最久的那个连接
 if (idleDurationNs > longestIdleDurationNs) {
 longestIdleDurationNs = idleDurationNs;
 longestIdleConnection = connection; }
 //空闲最久的那个连接 //如果空闲时间大于keepAliveDurationNs
 (默认5分钟) //或者空闲的连接总数大于maxIdleConnections
 (默认5个) //--->执行移除操作 if
 (longestIdleDurationNs >= this.keepAliveDurationNs
 || idleConnectionCount > this.maxIdleConnections) {
 // We've found a connection to evict. Remove it from
 the list, then close it below (outside // of the
 synchronized block).
 connections.remove(longestIdleConnection); } else if
 (idleConnectionCount > 0) { // A connection will
 be ready to evict soon. return keepAliveDurationNs
 - longestIdleDurationNs; //空闲最久的那个连接的空闲时长与
 keepAliveDurationNs的差值 } else if
 (inUseConnectionCount > 0) { // All connections
 are in use. It'll be at least the keep alive duration
 'til we run again. return keepAliveDurationNs;
} else { // No connections, idle or in use.
cleanupRunning = false; return -1; } }
```

```
closeQuietly(longestIdleConnection.socket()); //关闭Socket
// Cleanup again immediately. return 0; } ````
```

这个方法根据两个指标还决定是否移除空闲时间最长的空闲连接：大于最大空闲值或者空闲连接数超过最大值，则移除空闲时间最长的控线连接。cleanUp方法的执行也依赖于另外一个比较重要的方法：pruneAndGetAllocationCount，该方法的作用是移除发生泄漏的StreamAllocation，统计连接中正在使用的StreamAllocation个数。这个方法的源码不看了，有兴趣的自行品尝吧。

b) 取

```
RealConnection get(Address address, StreamAllocation
streamAllocation, Route route) { assert
(Thread.holdsLock(this)); for (RealConnection
connection : connections) { //isEligible判断一个连接
(address+route对应的) // 是否还能携带一个
streamAllocation。如果有，说明这个连接可用 if
(connection.isEligible(address, route)) { //isEligible也是一个
重要方法，最好看一下源码
streamAllocation.acquire(connection, true); //将
streamAllocation添加到connection.allocations中
return connection; } } return null; }
```

首先，判断address对应的Connection是否还能承载一个新的StreamAllocation，如果可以得话，我们就将这个streamAllocation添加到connection.allocations中。最后返回这个Connection。

c) 移除

```
``` public void evictAll() {     List<RealConnection>  
evictedConnections = new ArrayList<>();     synchronized  
(this) {         for (Iterator<RealConnection> i =  
connections.iterator(); i.hasNext(); ) {  
RealConnection connection = i.next();             if  
(connection.allocations.isEmpty()) {  
connection.noNewStreams = true;  
evictedConnections.add(connection);             i.remove();  
}     }     for (RealConnection connection :  
evictedConnections) {  
closeQuietly(connection.socket());     } } ```
```

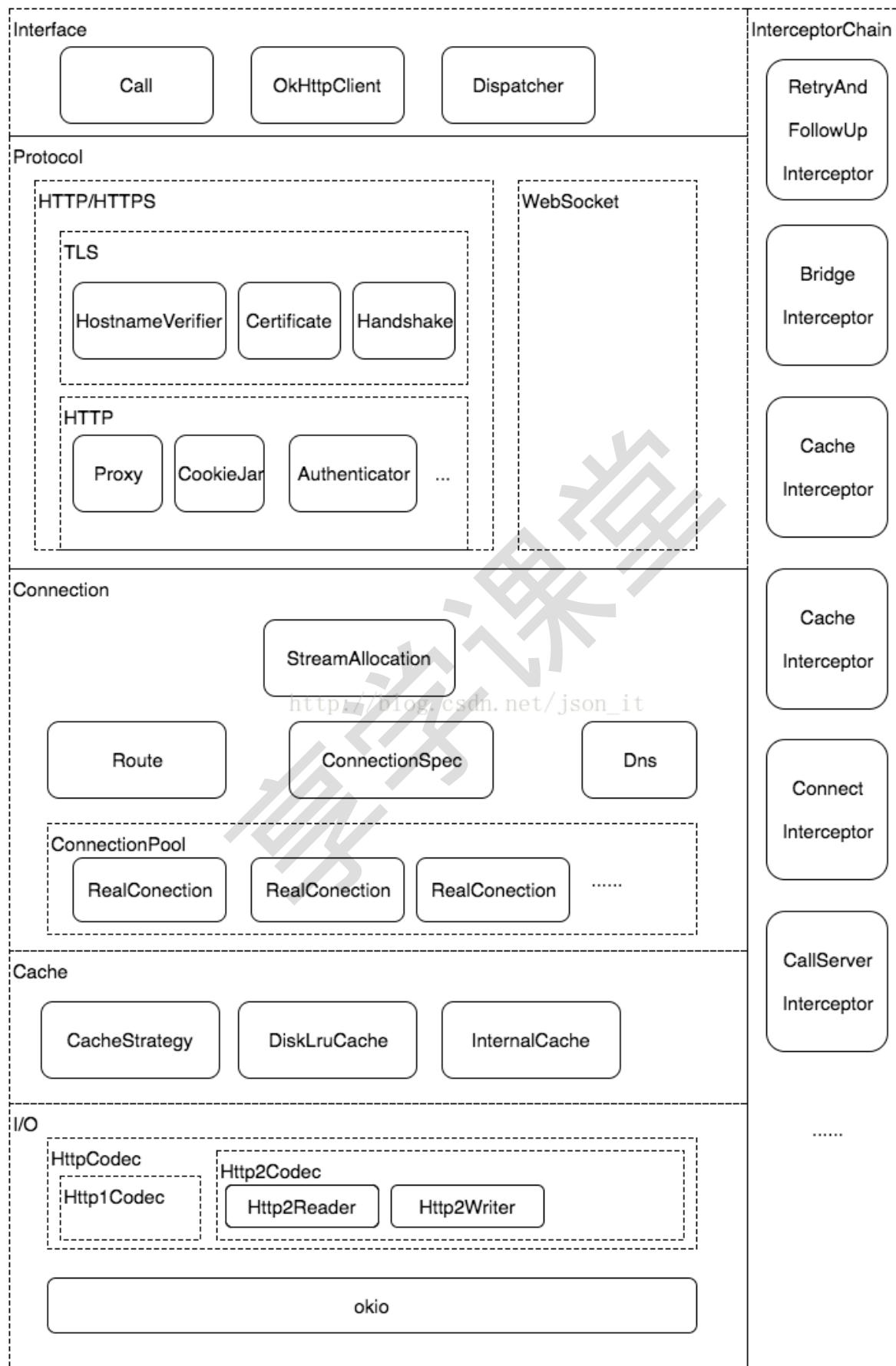
这个就很简单了。不再叙述。

3.4.5、CallServerInterceptor

该拦截器就是利用HttpCodec完成最终请求的发送。

4、总结

okhttp是一个Http+Http2客户端，适用于Android + Java 应用。其整体的架构如下：（此图来源于<https://yq.aliyun.com/articles/78105?spm=5176.100239.blogcont78104.10.FIPFWr>，感谢）



整体分析完之后，再看这个整体架构，感觉上图画的十分清晰。云栖社区的这篇 [《OkHttp 3.7源码分析》](#) 写的相当不错。有时间的可以阅读以下。

8.4.3 Android okhttp3 创建Socket的底层实现追踪

(这篇描述了OKhttp3 socket底层的实现)

1.概述

OkHttp3的最底层是Socket，而不是URLConnection，它通过Platform的Class.forName()反射获得当前Runtime使用的socket库，调用栈如下

okhttp//实现HTTP协议

==>framework//JRE，实现JDK中Socket封装

==>jvm//JDK的实现，本质对libc标准库的native封装

==>bionic//android下的libc标准库

==>systemcall//用户态切换入内核

==>kernel//实现下协议栈(L4,L3)与网络驱动(一般是L2,L1)

注：需求决定，Android版本4.4.4 okhttp 3.2.0

2.因为底层使用Socket，所以在okhttp3源码全局搜索"new Socket"这个关键词，定位在：

okhttp3.internal.io.RealConnection#connect

```
rawSocket = proxy.type() == Proxy.Type.DIRECT ||
proxy.type() == Proxy.Type.HTTP
    ? address.socketFactory().createSocket()
    : new Socket(proxy);
```

3.在此处打断点，调试

```
<init>:75, PlainSocketImpl (java.net)
<init>:57, Socket (java.net)
createSocket:35, DefaultSocketFactory (javax.net)
connect:109, RealConnection (okhttp3.internal.io)
findConnection:188, StreamAllocation (okhttp3.internal.http)
findHealthyConnection:127, StreamAllocation (okhttp3.internal.http)
newStream:97, StreamAllocation (okhttp3.internal.http)
connect:289, HttpEngine (okhttp3.internal.http) http://blog.csdn.net/
sendRequest:241, HttpEngine (okhttp3.internal.http)
getResponse:240, RealCall (okhttp3)
proceed:198, RealCall$ApplicationInterceptorChain (okhttp3)
intercept:52, OkHttp3Interceptor (com.baidu.uaq.agent.android.instrumentation.okhttp3util)
proceed:187, RealCall$ApplicationInterceptorChain (okhttp3)
getResponseBodyWithInterceptorChain:160, RealCall (okhttp3)
execute:57, RealCall (okhttp3)
```

rawSocket为

```
▼ rawSocket = {Socket@830041057704} "Socket[unconnected]"
  ► f connectLock = {Object@830041057760}
  ▼ f impl = {PlainSocketImpl@830041057776} "Socket[address=null,port=0,localPort=0]"
    ► f guard = {CloseGuard@830027937024}
      f proxy = null
      f shutdownInput = false
      f streaming = true
      f address = null
    ▼ f fd = {FileDescriptor@830041057824} "FileDescriptor[-1]"
      f descriptor = -1
      f localport = 0
      f port = 0
      f proxy = null
    ▼ f localAddress = {InetAddress@830036886000} "/0.0.0.0"
      ► f ipaddress = {byte[4]@830036886024}
      f hostName = null
      f family = 2
      f isConnected = false
      f isCreated = false
      f isInputShutdown = false
      f isOutputShutdown = false
      f isClosed = false
      f isBound = false
http://blog.csdn.net/
```

所以address.socketFactory()返回的是DefaultSocketFactory.java

4./libcore/luni/src/main/java/javax/net/DefaultSocketFactory.java

- ```
 /**
 * Default implementation of {@link
 javax.net.SocketFactory}
 */
final class DefaultSocketFactory extends
SocketFactory {

 DefaultSocketFactory() {
 }

 @Override
 public Socket createSocket() throws IOException {
 return new Socket();
 }
}
```

#### 5./ libcore/ luni/ src/ main/ java/ java/ net/ Socket.java

```
public Socket() {
 this.impl = factory != null ?
factory.createSocketImpl() : new PlainSocketImpl();
 this.proxy = null;
}
```

#### 6./ libcore/ luni/ src/ main/ java/ java/ net/ PlainSocketImpl.java

```
public PlainSocketImpl() {
 this(new FileDescriptor());
}
```

#### 7.以上步骤还没通过jni调用libc.so，真正创建socket是在connectSocket中

#### okhttp3.internal.io.RealConnection#connect

```
connectSocket(connectTimeout, readTimeout, writeTimeout,
connectionSpecSelector);
```

#### 8.okhttp3.internal.io.RealConnection#connectSocket

```
/** Does all the work necessary to build a full HTTP or
HTTPS connection on a raw socket. */
private void connectSocket(int connectTimeout, int
readTimeout, int writeTimeout,
 ConnectionSpecSelector connectionSpecSelector)
throws IOException {
 rawSocket.setSoTimeout(readTimeout);
```

9./ libcore/ luni/ src/ main/ java/ java/ net/ Socket.java#setSoTimeout

```
/**
 * Sets this socket's {@link SocketOptions#SO_TIMEOUT}
read timeout} in milliseconds.
 * Use 0 for no timeout.
 * To take effect, this option must be set before the
blocking method was called.
 */
public synchronized void setSoTimeout(int timeout)
throws SocketException {
 checkOpenAndCreate(true);
 if (timeout < 0) {
 throw new IllegalArgumentException("timeout <
0");
 }
 impl.setOption(SocketOptions.SO_TIMEOUT,
Integer.valueOf(timeout));
}
```

10./ libcore/ luni/ src/ main/ java/ java/ net/

```
Socket.java#checkOpenAndCreate
/**
 * Checks whether the socket is closed, and throws an
exception. Otherwise
 * creates the underlying SocketImpl.
 *
 * @throws SocketException
 * if the socket is closed.
```

```
 */
private void checkOpenAndCreate(boolean create)
throws SocketException {
 if (isClosed()) {
 throw new SocketException("Socket is
closed");
 }
 if (!create) {
 if (!isConnected()) {
 throw new SocketException("Socket is not
connected");
 }
 // a connected socket must be created
 }
}
```

```
/*
 * return directly to fix a possible bug, if
!create, should return
 * here
 */
return;
}
if (isCreated) {
 return;
}
synchronized (this) {
 if (isCreated) {
 return;
 }
 try {
 impl.create(true);
 } catch (SocketException e) {
 throw e;
 } catch (IOException e) {
 throw new SocketException(e.toString());
 }
 isCreated = true;
}
```

最后调用PlainSocketImpl.java的create

10./libcore/luni/src/main/java/java/net/PlainSocketImpl.java

```
protected void create(boolean streaming) throws
IOException {
 this.streaming = streaming;
 this.fd = IoBridge.socket(streaming);
}
```

11./ libcore/ luni/ src/ main/ java/ libcore/ io/ IoBridge.java

```
public static FileDescriptor socket(boolean stream)
throws SocketException {
 FileDescriptor fd;
 try {
 fd = Libcore.os.socket(AF_INET6, stream ?
SOCK_STREAM : SOCK_DGRAM, 0);

 // The RFC (http://www.ietf.org/rfc/rfc3493.txt)
 // says that IPV6_MULTICAST_HOPS defaults // to 1.
 // The Linux kernel (at least up to 2.6.38) accidentally
 // defaults to 64 (which // would be correct for the
 // *unicast* hop limit). // See
 // http://www.spinics.net/lists/netdev/msg129022.html,
 // though no patch appears to // have been applied as
 // a result of that discussion. If that bug is ever fixed,
 // we can // remove this code. Until then, we
 // manually set the hop limit on IPV6 datagram sockets.
 // (IPv4 is already correct.) if (!stream) {
 Libcore.os.setsockoptInt(fd, IPPROTO_IPV6,
IPV6_MULTICAST_HOPS, 1); } return fd; }
 catch (ErrnoException errnoException) { throw
 errnoException.rethrowAsSocketException(); }}
```

总算往jni方向去了： Libcore.os.socket(AF\_INET6, stream ?  
SOCK\_STREAM : SOCK\_DGRAM, 0)  
可见： 创建socket时， domain为AF\_INET6， 类型为SOCK\_STREAM (对于http来说)

在c层可以用这两个条件来过滤http的socket创建

12.

/libcore/luni/src/main/java/libcore/io/BlockGuardOs.java

```
public FileDescriptor socket(int domain, int type, int protocol) throws ErrnoException { return tagSocket(os.socket(domain, type, protocol)); }
```

/libcore/luni/src/main/java/libcore/io/ForwardingOs.java

```
public FileDescriptor socket(int domain, int type, int protocol) throws ErrnoException { return os.socket(domain, type, protocol); }
```

13./ libcore/ luni/ src/ main/ java/ libcore/ io/ Posix.java

```
public native FileDescriptor socket(int domain, int type, int protocol) throws ErrnoException;
```

14./ libcore / luni / src / main / native / libcore\_io\_Posix.cpp     JNI  
层，此方法为java的代理

```
static jobject Posix_socket(JNIEnv* env, jobject, jint domain, jint type, jint protocol) {int fd = throwIfMinusOne(env, "socket", TEMP_FAILURE_RETRY(socket(domain, type, protocol)));return fd != -1 ? jniCreateFileDescriptor(env, fd) : NULL;}
```

```
NATIVE_METHOD(Posix, socket, "(III)Ljava/io/FileDescriptor;") ,
```

15./bionic/libc/arch-arm/syscalls/socket.S

socket通过汇编实现，汇编代码中通过swi调用中断号实现功能

```
ENTRY(socket) mov ip, r7 ldr r7,
=__NR_socket swi #0 mov r7, ip cmn
r0, #(MAX_ERRNO + 1) bxls lr neg r0, r0 b
 __set_errnoEND(socket)
```

至此，socket创建跟踪完毕~

#### 8.4.4 OkHttp3源码和设计模式：

(这篇从设计模式的角度分析OKhttp3)

在[《打车APP实战》](#)课程中，我们使用OkHttp3简单搭建了一个网络框架，实践了OkHttp3的用法。不过课程本身的重点是MVP架构的实践，所以没有进一步OkHttp3底层的实现细节。本文来探究一下OkHttp3的源码和其中的设计思想。

关于OkHttp3的源码分析的文章挺多，不过大多还是在为了源码而源码。个人觉得如果读源码不去分析源码背后的设计模式或设计思想，那么读源码的意义不大。同时，如果熟悉的设计模式越多，那么读某个框架的源码的时候就越容易，两者是相辅相成的，这也是许多大牛认为多读源码能提高编程能力的原因。

#### 整体架构

为了方便后面的理解，我这里简单画了个架构图，图中画出了OkHttp3核心的功能模块。为了方便整体理解，这里分了三个层次：客户层、执行层和连接层。

首先，客户层的OkHttpClient，使用过OkHttp网络库的同学应该都熟悉，在发送网络请求，执行层决定怎么处理请求，比如同步还是异步，同步请求的话直接在当前线程完成请求，请求要经过多层拦截器处理；如果是异步处理，需要Dispatcher执行分发策略，线程池管理执行任务；又比如，一个请求下来，要不要走缓存，如果不走缓存，进行网络请求。最后执行层将从连接层进行网络IO获取数据。

#### OkHttpClient

使用过 OkHttp 网络库的同学应该都熟悉 OkHttpClient，许多第三方框架都会提供一个类似的类作为客户访问的一个入口。关于 OkHttpClient 代码注释上就说的很清楚：

```
/** * Factory for {@linkplain call calls}, which can
be used to send HTTP requests and read their *
responses. * * <h3>okHttpClients should be shared</h3> *
* <p>OkHttp performs best when you create a single {@code
OkHttpClient} instance and reuse it for * all of your
HTTP calls. This is because each client holds its own
connection pool and thread * pools. Reusing connections
and threads reduces latency and saves memory. Conversely,
creating a * client for each request wastes resources on
idle pools. * * <p>Use {@code new OkHttpClient()} to
create a shared instance with the default settings: *
<pre> {@code * * // The singleton HTTP client. *
public final OkHttpClient client = new OkHttpClient(); *
}</pre> * * <p>Or use {@code new OkHttpClient.Builder()}
to create a shared instance with custom settings: *
<pre> {@code * * // The singleton HTTP client. *
public final OkHttpClient client = new
OkHttpClient.Builder() * .addInterceptor(new
HttpLoggingInterceptor()) * .cache(new
Cache(cacheDir, cacheSize)) * .build(); * }</pre> *
.... 省略*/
```

简单提炼：

1、 OkHttpClient，可以通过 new OkHttpClient() 或 new OkHttpClient.Builder() 来创建对象，但是---特别注意， OkHttpClient() 对象最好是共享的，建议使用单例模式创建。因为每个 OkHttpClient 对象都管理自己独有的线程池和连接池。这一点很多同学，甚至在我经历的团队中就有人踩过坑，每一个请求都创建一个 OkHttpClient 导致内存爆掉。

2、从上面的整体框架图，其实执行层有很多属性功能是需要 OkHttpClient 来制定，例如缓存、线程池、拦截器等。如果你是设计者你会怎样设计 OkHttpClient？建造者模式，OkHttpClient 比较复杂，太多属性，而且客户的组合需求多样化，这种情况下就考虑使用建造者模式。new OkHttpClient() 创建对象，内部默认指定了很多属性：

```
public OkHttpClient() { this(new Builder());}
```

在看看 new Builder() 的默认实现：

```
public Builder() { dispatcher = new Dispatcher();
protocols = DEFAULT_PROTOCOLS; connectionSpecs =
DEFAULT_CONNECTION_SPECS; eventListenerFactory =
EventListener.factory(EventListener.NONE); proxySelector
= ProxySelector.getDefault(); cookieJar =
CookieJar.NO_COOKIES; socketFactory =
SocketFactory.getDefault(); hostnameVerifier =
OkHostnameVerifier.INSTANCE; certificatePinner =
CertificatePinner.DEFAULT; proxyAuthenticator =
Authenticator.NONE; authenticator = Authenticator.NONE;
connectionPool = new ConnectionPool(); dns = Dns.SYSTEM;
followSslRedirects = true; followRedirects = true;
retryOnConnectionFailure = true; connectTimeout =
10_000; readTimeout = 10_000; writeTimeout = 10_000;
pingInterval = 0;}
```

默认指定 Dispatcher（管理线程池）、链接池、超时时间等。

3、内部对于线程池、链接池管理有默认的管理策略，例如空闲时候的线程池、连接池会在一定时间自动释放，但如果你想主动去释放也可以通过客户层去释放。（很少）

## 执行层

```
Response response =
mokHttpClient.newCall(request).execute();
```

这是应用程序中发起网络请求最顶端的调用，newCall(request) 方法返回 RealCall 对象。RealCall 封装了一个 request 代表一个请求调用任务，RealCall 有两个重要的方法 execute() 和 enqueue(Callback responseCallback)。execute() 是直接在当前线程执行请求，enqueue(Callback responseCallback) 是将当前任务加到任务队列中，执行异步请求。

## 同步请求

```
@Override public Response execute() throws IOException {
 synchronized (this) {
 if (executed) throw new
IllegalStateException("Already Executed");
 executed =
true; } captureCallStackTrace(); try { //
client.dispatcher().executed(this) 内部只是记录下执行状态，
client.dispatcher().executed(this); // 真正执行发生在这里
 Response result = getResponseWithInterceptorChain();
 if (result == null) throw new IOException("Canceled");
 return result; } finally { // 后面再解释
client.dispatcher().finished(this); }}}
```

执行方法关键在 getResponseWithInterceptorChain() 这个方法中，关于 client.dispatcher().executed(this) 和 client.dispatcher().finished(this); 这里先忽略，后面再看。

请求过程要从执行层说到连接层，涉及到 getResponseWithInterceptorChain 方法中组织的各个拦截器的执行过程，内容比较多，后面章节再说。先说说 RealCall 中 enqueue(Callback responseCallback) 方法涉及的异步请求和线程池。

## Dispatcher 和线程池

```
@Override public void enqueue(Callback responseCallback)
{ synchronized (this) { if (executed) throw new
IllegalStateException("Already Executed"); executed =
true;} captureCallStackTrace();
client.dispatcher().enqueue(new
AsyncCall(responseCallback));}}
```

调用了 dispatcher 的 enqueue()方法  
dispatcher 结合线程池完成了所有异步请求任务的调配。

```
synchronized void enqueue(AsyncCall call) {
 if (runningAsyncCalls.size() < maxRequests &&
 runningCallsForHost(call) < maxRequestsPerHost) {
 runningAsyncCalls.add(call);
 executorService().execute(call);
 } else {
 readyAsyncCalls.add(call);
 }
}
```

dispatcher 主要维护了三两个队列 readyAsyncCalls、runningAsyncCalls 和 runningSyncCalls，分别代表了准备中队列，正在执行的异步任务队列和正在执行的同步队列，重点关注下前面两个。  
现在我们可以回头来看看前面 RealCall 方法 client.dispatcher().finished(this) 这个疑点了。在每个任务执行完之后要回调 client.dispatcher().finished(this) 方法，主要是要将当前任务从 runningAsyncCalls 或 runningSyncCalls 中移除，同时把 readyAsyncCalls 的任务调度到 runningAsyncCalls 中并执行。

## 线程池

```
public synchronized ExecutorService executorService() {
 if (executorService == null) {
 executorService = new
 ThreadPoolExecutor(0, Integer.MAX_VALUE, 60,
 TimeUnit.SECONDS, new SynchronousQueue<Runnable>(),
 util.threadFactory("OkHttp Dispatcher", false));
 }
 return executorService; }
```

默认实现是一个不限容量的线程池，线程空闲时存活时间为 60 秒。线程池实现了对象复用，降低线程创建开销，从设计模式上来讲，使用了享元模式。

## 责任链（拦截器执行过程）

```
Response getResponseWithInterceptorChain() throws
IOException { // Build a full stack of
interceptors.List<Interceptor> interceptors = new
ArrayList<>
(); interceptors.addAll(client.interceptors()); interceptor
s.add(retryAndFollowUpInterceptor); interceptors.add(new
BridgeInterceptor(client.cookieJar())); interceptors.add(n
ew
CacheInterceptor(client.internalCache())); interceptors.ad
d(new ConnectInterceptor(client)); if (!forWebSocket) {
interceptors.addAll(client.networkInterceptors()); } interc
eptors.add(new
callServerInterceptor(forWebSocket)); Interceptor.Chain
chain = new RealInterceptorChain(interceptors, null,
null, null, 0, originalRequest); return
chain.proceed(originalRequest); }}
```

要跟踪 Okhttp3 的网络请求任务执行过程，需要看懂以上代码，看懂以上代码必须理解设计模式-责任链。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。网络请求过程，是比较典型的复合责任链的场景，比如请求传递过程，我们需要做请求重试，需要执行缓存策略，需要建立连接等，每一个处理节点可以由一个链上的对象来处理；同时客户端使用的时候可能也会在请求过程中做一些应用层需要的事情，比如我要记录网络请求的耗时、日志等，责任链还可以动态的扩展到客户业务方。

在 OkHttp3 的拦截器链中，内置了5个默认的拦截器，分别用于重试、请求对象转换、缓存、链接、网络读写。

以上方法中先是添加了客户端自定义的连接器，然后在分别添加内置拦截器。

## Okhttp3 拦截器类图

现在我们把对 OkHttp 网络请求执行过程的研究转化对每个拦截器处理的研究。

## retryAndFollowUpInterceptor 重试机制

retryAndFollowUpInterceptor 处于内置拦截器链的最顶端，在一个循环中执行重试过程：

- 1、首先下游拦截器在处理网络请求过程如抛出异常，则通过一定的机制判断一下当前链接是否可恢复的（例如，异常是不是致命的、有没有更多的线路可以尝试等），如果可恢复则重试，否则跳出循环。
- 2、如果没什么异常则校验下返回状态、代理鉴权、重定向等，如果需要重定向则继续，否则直接跳出循环返回结果。
- 3、如果重定向，则要判断下是否已经达到最大可重定向次数，达到则抛出异常，跳出循环。

```
@Override public Response intercept(Chain chain) throws
IOException {
 Request request = chain.request();
 // 创建连接池管理对象
 StreamAllocation streamAllocation = new StreamAllocation(
 client.connectionPool(), createAddress(request.url()), callStackTrace);
```

```
int followUpCount = 0; Response priorResponse = null; while (true) { if (canceled) { streamAllocation.release(); throw new IOException("Canceled"); } Response response = null; boolean releaseConnection = true; try { // 将请求处理传递下游拦截器处理 response = ((RealInterceptorChain) chain).proceed(request, streamAllocation, null, null); releaseConnection = false; } catch (RouteException e) { // The attempt to connect via a route failed. The request will not have been sent. // 线路异常，判断满足可恢复条件，满足则继续循环重试 if (!recover(e.getLastConnectException(), false, request)) { throw e.getLastConnectException(); } releaseConnection = false; continue; } catch (IOException e) { // An attempt to communicate with a server failed. The request may have been sent.
```

```
// IO异常，判断满足可恢复条件，满足则继续循环重试
boolean requestSendStarted = !(e instanceof ConnectionShutdownException);
if (!recover(e, requestSendStarted, request)) throw e;
releaseConnection = false;
continue;
} finally {
// We're throwing an unchecked exception. Release any resources.
if (releaseConnection) {
streamAllocation.streamFailed(null);
streamAllocation.release();
}
}
```

```
// Attach the prior response if it exists. Such
responses never have a body. if (priorResponse != null)
{ response = response.newBuilder()
.priorResponse(priorResponse.newBuilder()
.body(null) .build()) .build(); }
// 是否需要重定向 Request followUp =
followUpRequest(response); if (followUp == null) { if
(!forWebSocket) { streamAllocation.release(); }
// 不需要重定向，正常返回结果 return response; }
closeQuietly(response.body()); if (++followUpCount >
MAX_FOLLOW_UPS) { // 达到次数限制
streamAllocation.release(); throw new
ProtocolException("Too many follow-up requests: " +
followUpCount); } if (followUp.body() instanceof
UnrepeatableRequestBody) { streamAllocation.release();
throw new HttpRetryException("Cannot retry streamed
HTTP body", response.code()); } if
(!sameConnection(response, followUp.url())) {
streamAllocation.release(); streamAllocation = new
StreamAllocation(client.connectionPool(),
createAddress(followUp.url()), callStackTrace); } else
if (streamAllocation.codec() != null) { throw new
IllegalStateException("Closing the body of " + response
+ " didn't close its backing stream. Bad
interceptor?"); } request = followUp; priorResponse =
response; }}
```

## BridgeInterceptor

```
/** * Bridges from application code to network code.
First it builds a network request from a user * request.
Then it proceeds to call the network. Finally it builds a
user response from the network * response. */
```

这个拦截器比较简单，一个实现应用层和网络层直接的数据格式编码的桥。第一：把应用层客户端传过来的请求对象转换为Http网络协议所需字段的请求对象。第二，把下游网络请求结果转换为应用层客户所需要的响应对象。这个设计思想来自适配器设计模式，大家可以去体会一下。

## CacheInterceptor 数据策略 (策略模式)

CacheInterceptor 实现了数据的选择策略，来自网络还是来自本地？这个场景也是比较契合策略模式场景，CacheInterceptor 需要一个策略提供者提供它一个策略（锦囊），CacheInterceptor 根据这个策略去选择走网络数据还是本地缓存。

缓存的策略过程：

- 1、请求头包含 "If-Modified-Since" 或 "If-None-Match" 暂时不走缓存
- 2、客户端通过 cacheControl 指定了无缓存，不走缓存
- 3、客户端通过 cacheControl 指定了缓存，则看缓存过期时间，符合要求走缓存。
- 4、如果走了网络请求，响应状态码为 304（只有客户端请求头包含 "If-Modified-Since" 或 "If-None-Match"，服务器数据没变化的话会返回 304 状态码，不会返回响应内容），表示客户端继续用缓存。

```
@Override public Response intercept(Chain chain) throws
IOException {
 Response cacheCandidate = cache != null
 ? cache.get(chain.request())
 : null;
 long now = System.currentTimeMillis();
 CacheStrategy strategy = new CacheStrategy.Factory(now,
 chain.request(), cacheCandidate).get();
 // 获取缓存策略
 Request networkRequest = strategy.networkRequest;
 Response cacheResponse = strategy.cacheResponse;
 if (cache != null) {
 cache.trackResponse(strategy);
 }
 if (cacheCandidate != null && cacheResponse == null) {
 closeQuietly(cacheCandidate.body()); // The cache candidate wasn't
 applicable. Close it.
 }
 // If we're forbidden from using the network and the cache is
 insufficient, fail.
```

```
if (networkRequest == null && cacheResponse == null) {
 return new Response.Builder()
 .request(chain.request())
 .protocol(Protocol.HTTP_1_1)
 .code(504)
 .message("Unsatisfiable Request (only-if-cached)")
 .body(Util.EMPTY_RESPONSE)
 .sentRequestAtMillis(-1L)
 .receivedResponseAtMillis(System.currentTimeMillis())
 .build();
}

// 走缓存
if (networkRequest == null) {
 return cacheResponse.newBuilder()
 .cacheResponse(stripBody(cacheResponse))
 .build();
}

Response networkResponse = null;
try {
 // 执行网络
 networkResponse = chain.proceed(networkRequest);
} finally {
 // If we're crashing on I/O or otherwise, don't leak the cache body.
 if (networkResponse == null && cacheCandidate != null) {
 closeQuietly(cacheCandidate.body());
 }
}
```

```
// 返回 304 仍然走本地缓存
if (cacheResponse != null) { if
(networkResponse.code() == HTTP_NOT_MODIFIED) {
Response response = cacheResponse.newBuilder()
.headers(combine(cacheResponse.headers(),
networkResponse.headers()))
.sentRequestAtMillis(networkResponse.sentRequestAtMillis())
.receivedResponseAtMillis(networkResponse.receivedRespon
eAtMillis())
.cacheResponse(stripBody(cacheResponse))
.networkResponse(stripBody(networkResponse))
.build(); networkResponse.body().close(); // update
the cache after combining headers but before stripping
the // Content-Encoding header (as performed by
initContentStream()).
cache.trackConditionalCacheHit();
cache.update(cacheResponse, response); return
response; } else {
closeQuietly(cacheResponse.body()); }}Response response
= networkResponse.newBuilder()
.cacheResponse(stripBody(cacheResponse))
.networkResponse(stripBody(networkResponse))
.build();if (cache != null) { if
(HttpHeaders.hasBody(response) &&
CacheStrategy.isCacheable(response, networkRequest)) {
// 存储缓存 CacheRequest cacheRequest =
cache.put(response); return
cachewritingResponse(cacheRequest, response); } if
(Method.invalidatesCache(networkRequest.method())) {
try { cache.remove(networkRequest); } catch
(IOException ignored) { // The cache cannot be
written. } }}return response;}
```

## 缓存实现

OkHttp3 内部缓存默认实现是使用的 DiskLruCache，这部分代码有点绕：

```
interceptors.add(new CacheInterceptor(client.internalCache()));
```

初始化 CacheInterceptor 时候 client.internalCache() 这里获取 OkHttpClient的缓存。

```
InternalCache internalCache() { return cache != null ?
 cache.internalCache : internalCache; }
```

注意到，这个方法是非公开的。客户端只能通过 OkHttpClient.Builder 的 cache(cache) 定义缓存，cache 是一个 Cache 对实例。在看看 Cache 的内部实现，内部有一个 InternalCache 的内部类实现。内部调用时使用 InternalCache 实例提供接口，而存储逻辑在 Cache 中实现。

Cache 为什么不直接实现 InternalCache，而通过持有 InternalCache 的一个内部类对象来实现方法？是希望控制缓存实现，不希望用户外部去实现缓存，同时对内保持一定的扩展。

## 链接层

RealCall 封装了请求过程，组织了用户和内置拦截器，其中内置拦截器 retryAndFollowUpInterceptor -> BridgeInterceptor -> CacheInterceptor 完成执行层的大部分逻辑，ConnectInterceptor -> CallServerInterceptor 两个拦截器开始迈向连接层最终完成网络请求。关于 ConnectInterceptor -> CallServerInterceptor 要结合连接层一起说明，限于篇幅，下一篇篇文章：《OkHttp3源码和设计模式-2》接着分析。

### 8.4.5 OkHttp3对RealCall的注释及上篇是Dispatcher

1. Note: **Call** 和 **Request** 在本系列文章中，均被称为请求。
2. 前者为OkHttp的请求（或者叫做命令），
3. 后者为HTTP的请求的封装类。请读者自行区分。

## 概述

通常，我们需要发送一个请求，那么，我们会先使用创建一个 Request，然后使用okHttpClient.newCall(request)，创建一个Call，然后使用这个Call，发出（同步 / 异步）请求，没错，这个Call，就是RealCall，不

多说，下面开始看源码。

## 源码

先看RealCall的类结构

真实的课时

## RealCall.java

```
RealCall.java
Show inherited members (⌘F12) Show Anonymous Classes (⌘I) ⚙

▼ C RealCall
 RealCall(OkHttpClient, Request)
 cancel(): void ↑Call
 enqueue(Callback): void ↑Call
 enqueue(Callback, boolean): void
 execute(): Response ↑Call
 getResponse(Request, boolean): Response
 getResponseWithInterceptorChain(boolean): Response
 isCanceled(): boolean ↑Call
 isExecuted(): boolean ↑Call
 request(): Request ↑Call
 tag(): Object
 toLoggableString(): String
 canceled: boolean
 client: OkHttpClient
 engine: HttpEngine
 executed: boolean
 originalRequest: Request

▼ C ApplicationInterceptorChain
 ApplicationInterceptorChain(int, Request, boolean)
 connection(): Connection ↑Chain
 proceed(Request): Response ↑Chain
 request(): Request ↑Chain
 forWebSocket: boolean
 index: int
 request: Request

▼ C AsyncCall
 AsyncCall(Callback, boolean)
 cancel(): void
 execute(): void ↑NamedRunnable
 get(): RealCall
 host(): String
 request(): Request
 tag(): Object
 forWebSocket: boolean
 responseCallback: Callback
```

RealCall为接口Call的实现类。

```

/** * 一个Call封装一对Request和Response，能且仅能被执行一次。并且Call可以被取消。 */
public interface Call { /* 返回初始化此Call的原始请求 */
 Request request(); /* *
立即发出请求，一直阻塞到响应可以被处理，或者发生了错误。 */
 /* *
可以调用Response#body方法获取到相应的body。为了连接服用，调用者需要调用ResponseBody#close()来关闭响应体。 */
 /* *
注意：传输层成功（收到响应码，响应头，响应体），不代表应用层成功。依然可能有404，或者500这些坑爹的响应码。 */
 /* @throws IOException 如果一个请求因为被取消、连接问题、超时，那么抛出此异常。可能是服务器在发生错误之前接收到了请求，造成网络在交互过程中出错。 */
 /* @throws IllegalStateException 当一个请求已经被执行，抛出此异常 */
 Response execute() throws IOException; /* *
安排请求在未来的某一刻执行。 */
 /* *
OkHttpClient#dispatcher决定这个请求什么时候被执行：通常立即就被执行了，除非是目前有其它的请求被执行。 */
 /* *
根据上篇的分析，如果不能立即执行，会被移动到就绪队列中。 */
 /* *
稍后，responseCallback会被调用，可能是一个正常的HTTP返回，或者是一个失败的异常。 */
 /* *
@throws IllegalStateException 当一个请求已经被执行，抛出此异常 */
 void enqueue(CallBack responseCallback); /* *
取消请求，如果可能的话。如果请求已经有返回了，那么就不能被取消了。 */
 void cancel(); /* *
返回true，如果execute()或者enqueue(callback)被执行过了。执行请求两次会出错的，还是判断下好。 */
 boolean isExecuted(); /* *
返回true，如果这个请求被取消了。 */
 boolean isCanceled(); /* *
这是一个生成请求的工厂接口。 */
 interface Factory {
 // 根据一个Http请求生成一个OKHttp请求。
 Call newCall(Request request); }
}

```

再来看RealCall中，这几个方法是如何实现的。

详细的分析过程，都写在了注释当中。

```

/* *
构造器，原始的请求保存为成员变量 */
protected RealCall(OkHttpClient client, Request originalRequest) {
 this.client = client; this.originalRequest =
 originalRequest; }

```

```
/**直接返回原始的请求*/@Override public Request request() {
 return originalRequest;}
```

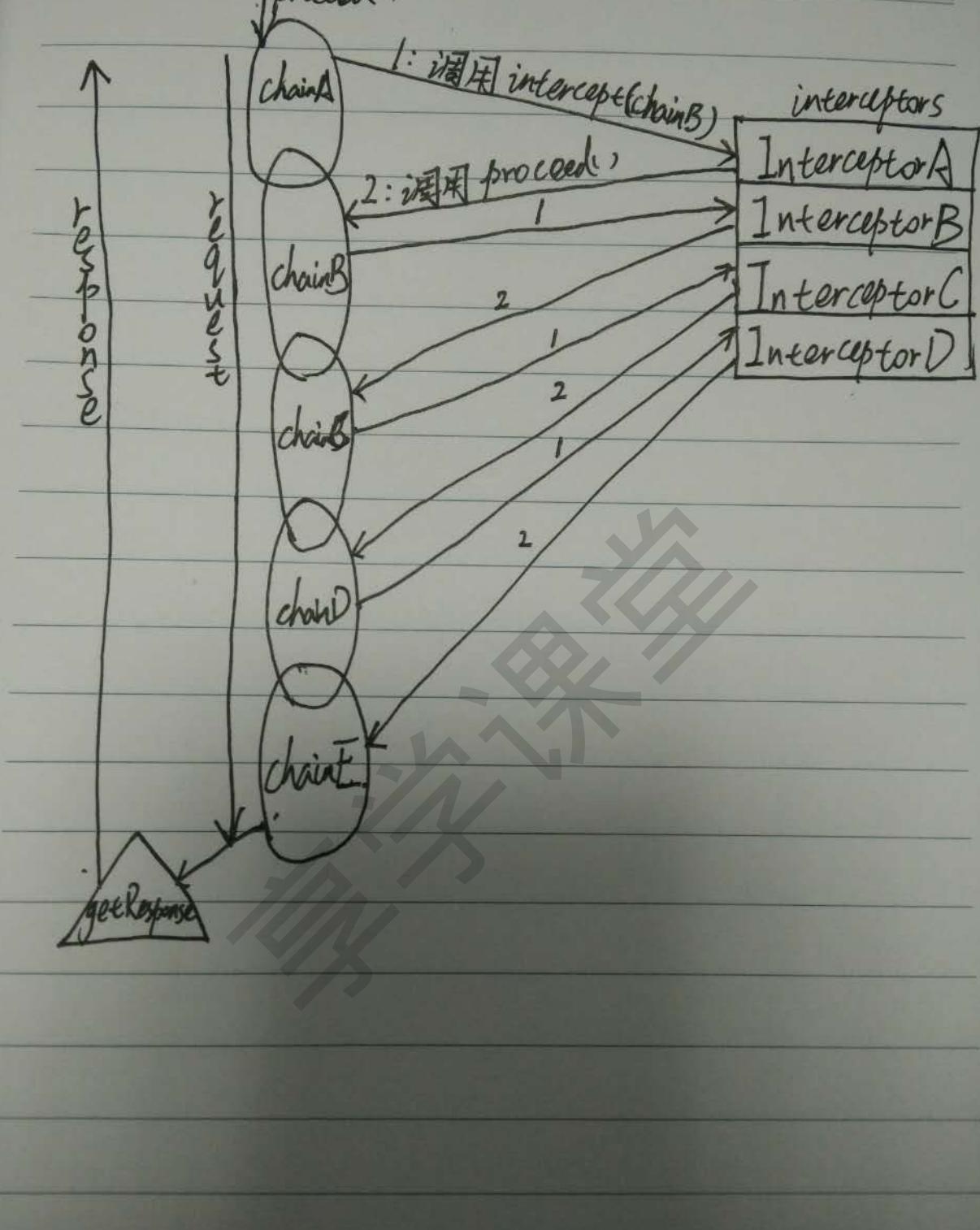
```
/**同步执行请求，会造成线程阻塞*/@Override public Response
execute() throws IOException { //如果已经被执行过了， 抛异常。
 synchronized (this) { if (executed) throw new
IllegalStateException("Already Executed"); //设置该请
求已经被执行。 executed = true; } try { //这
里调用的dispatcher的executed方法，将此请求加入到同步请求运行队列
中 client.dispatcher().executed(this); //下面的一
行是此方法的核心，组成连接器链，将这个请求加入到拦截器链中 //
在所有的拦截器都拦截一遍后，发送http请求，获取response //下
面会详细讲解这个方法 Response result =
getResponsewithInterceptorChain(false); if (result
== null) throw new IOException("Canceled"); return
result; } finally { //将运行中队列中的请求移除
client.dispatcher().finished(this); } }
```

```
/**进入到拦截器链中，并获取响应*/private Response
getResponsewithInterceptorChain(boolean forWebSocket)
throws IOException { // 新建一个
ApplicationInterceptorChain实例，请求作为构造器参数传入
Interceptor.Chain chain = new
ApplicationInterceptorChain(0, originalRequest,
forWebSocket); // 调用此方法，进入拦截器链 return
chain.proceed(originalRequest);}
```

## ApplicationInterceptorChain

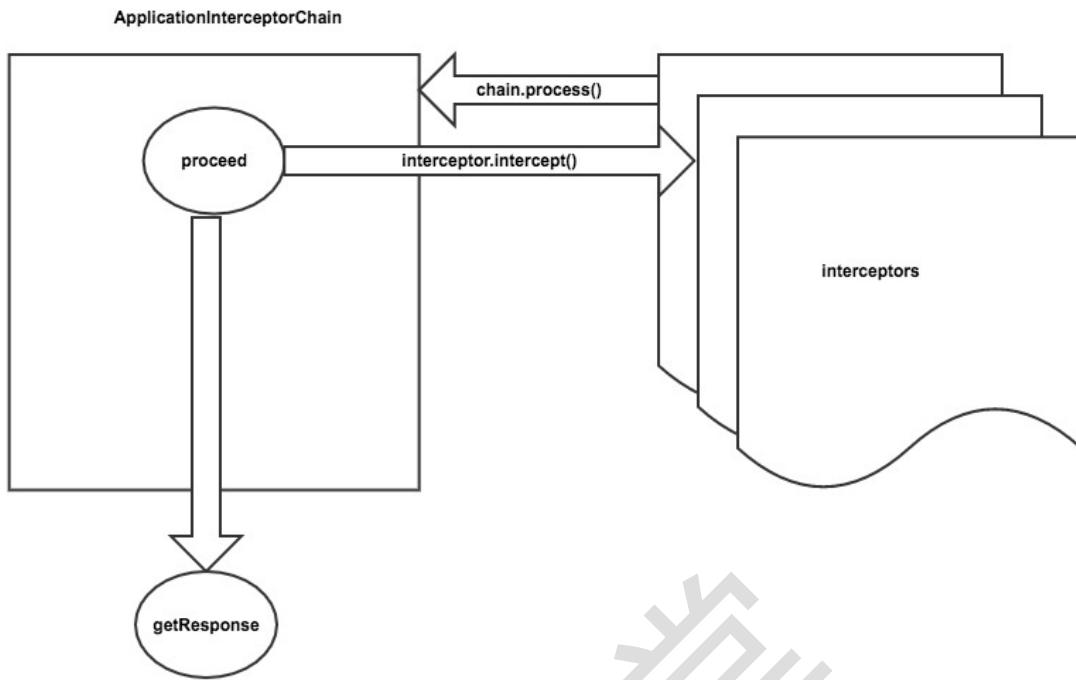
```
/**ApplicationInterceptorChain 的构造器index 为这一环节的索引
request 上一个环节拦截后的请求
*/ApplicationInterceptorChain(int index, Request request,
boolean forwebsocket) { this.index = index;
this.request = request; this.forwebsocket =
forwebsocket;}/*这个方法用于连接每个拦截器，并执行http请求除了进入拦截器链的时候，由RealCall#getResponsewithInterceptorChain方法调用，其它时刻都是在拦截器的Interceptor#intercept方法中，由用户主动调用此方法因为需要实现拦截器的功能，必须要调用
Interceptor#intercept，因此是用户参与了此拦截器链的建立
*/
@Override public Response proceed(Request request)
throws IOException { // 如果此环节的索引没有超过拦截器的大小，则再建立一个ApplicationInterceptorChain对象， // 传递给Interceptor的intercept方法，实现递归调用 if (index <
client.interceptors().size()) { // 新建一个环节，并且将索引+1，这里的request可能不再是originalRequest了，因为在拦截器中可能被修改了 Interceptor.Chain chain = new
ApplicationInterceptorChain(index + 1, request,
forwebsocket); // 获取到对应的拦截器
Interceptor interceptor =
client.interceptors().get(index); // 执行拦截器的
intercept方法，参数是上面新建的环节，这个方法里面会调用
chain.proceed(), 递归了。 // 在最深的一层调用
getResponse之后，响应会一层层的往外传 Response
interceptedResponse = interceptor.intercept(chain);```
if (interceptedResponse == null) { throw new
NullPointerException("application interceptor " +
interceptor + " returned null"); } // 返回
这一层拦截器处理用的响应 return intercepteResponse; }
// 递归到了最深的一层，拦截器都进入过了，发送httpRequest，获取到
response. // 这个方法相当复杂，因此放在最后讲解 return
getResponse(request, forwebsocket);}```
```

上面的代码讲解可能还未理解，下面祭出我的灵魂画作。



从`getResponseWithInterceptorChain()`进入到拦截器链中，然后请求被`ApplicationInterceptorChain`承载，传递到各个拦截器中，进入到所有的拦截器中后，调用`realCall.getResponse()`获取到响应，然后再一层层往上返回。

下面这个图的意思是一样的，看能不能更清楚点。



下面继续看RealCall的其它方法

```

/** 调用的是下面的重载的方法 */@Override public void
enqueue(Callback responseCallback) {
enqueue(responseCallback, false); }/**同样的，已经执行了，抛出
异常然后将此realCall包装为AsyncCall，调用Dispatcher#enqueue方
法。上一篇文章已经将结果Dispatcher了，这里就不再赘述。 */void
enqueue(Callback responseCallback, boolean forwebsocket)
{ //如果已经被执行过了， 抛异常。 synchronized (this) {
if (executed) throw new IllegalStateException("Already
Executed"); // 标识为已执行 executed = true; } //
AsyncCall是RealCall的内部类，相当于是对RealCall的一层封装，然后
将其传入就绪或运行队列中，等待线程池执行请求 // 下面会分析
AsyncCall client.dispatcher().enqueue(new
AsyncCall(responseCallback, forwebsocket)); }

```

## AsyncCall

```
/** AsyncCall的代码较短，因此直接贴出整个类 AsyncCall并不是 Call接口的实现类，而是继承自NamedRunnable抽象类，而这个父类，只是一个带设置线程名称的Runnable而已。 我们主要看execute()方法，这个方法是由Dispatcher中的线程池调用的。 */
final class AsyncCall
extends NamedRunnable {
 private final callback
 responseCallback; private final boolean
 forWebSocket;private AsyncCall(callback responseCallback,
boolean forWebSocket) { super("OkHttp %s",
originalRequest.url().toString()); this.responseCallback =
responseCallback; this.forWebSocket =
forWebSocket;}String host() { return
originalRequest.url().host();}Request request() { return
originalRequest;}Object tag() { return
originalRequest.tag();}void cancel() {
RealCall.this.cancel();}RealCall get() { return
RealCall.this;}
```

```
/** 过一遍拦截器链，并执行请求，然后调用回调函数。 */
protected void execute() { // 保证 onFailure 最多只会被调用一次
boolean signalledCallback = false; try { // 进入连接器链，并执行请求
Response response =
getResponseWithInterceptorChain(forWebSocket); // 如果请求被取消，调用 onFailure
if (canceled) {
signalledCallback = true;
responseCallback.onFailure(RealCall.this, new
IOException("Canceled")); } else { // 正常情况，调用 onResponse
signalledCallback = true;
responseCallback.onResponse(RealCall.this, response); }
} catch (IOException e) { // 如果上面有调用过回调，就不调了，这里保证 onFailure 只会被调用一次
if (signalledCallback) { logger.log(Level.INFO,
"Callback failure for " + toLoggableString(), e); }
else { responseCallback.onFailure(RealCall.this, e); }
} finally { // 运行队列移除请求
client.dispatcher().finished(this); } }
```

想  
識  
思  
學

```
/** * 执行请求，并获取响应结果。分多钟情况： 1. 请求被取消，抛出异常 2. 需要重定向，如果没有超过最大次数，重现创建HttpEngine，再次发出请求；如果超过最大次数，抛出异常 3. 连接出错，尝试恢复HttpEngine，再次发出请求；恢复失败，抛出异常 4. 发送的请求有问题，直接抛出异常 */ Response
getResponse(Request request, boolean forWebSocket) throws IOException {` ` ` // 复制请求头，并设置适合的属性。如果长度不为-1，则设置Content-Length，否则使用chunked方式传输。// 如果对chunked不熟悉，请参考其他资料RequestBody body =
request.body();if (body != null) { // 根据传进来的request创建新的Builder. Request.Builder requestBuilder =
request.newBuilder(); // 设置Content-Type MediaType
contentType = body.contentType(); if (contentType !=
null) { requestBuilder.header("Content-Type",
contentType.toString()); } // 判断使用何种方式传输 long
contentLength = body.contentLength(); // body长度不为-1，
设置Content-Length if (contentLength != -1) {
requestBuilder.header("Content-Length",
Long.toString(contentLength));
requestBuilder.removeHeader("Transfer-Encoding"); } else
{ // body 长度为 -1，使用chunked传输
requestBuilder.header("Transfer-Encoding", "chunked");
requestBuilder.removeHeader("Content-Length"); } // 创建新请求 request = requestBuilder.build();}// 创建一个新的引擎，每个引擎代表一次请求 / 响应对engine = new HttpEngine(client,
request, false, false, forWebSocket, null, null,
null);int followUpCount = 0; // 重试次数// 死循环 出口： // 1.
请求被取消// 2. 请求有问题// 3. 捕获到异常，且尝试恢复失败// 4.
获取到响应，且无需重定向// 5. 重定向次数超过最大限制while (true)
{ // 被取消的情况 if (canceled) {
engine.releaseStreamAllocation(); throw new
IOException("Canceled"); } boolean releaseConnection =
true; try { // 发送请求 engine.sendRequest(); //
读取响应 engine.readResponse(); releaseConnection =
false; } catch (RequestException e) { // 请求失败，请求
本身有问题，或者是网络不通 throw e.getCause(); } catch
(RouteException e) { // 连接到服务器的路由发生异常，请求还没
被发送 // 通过上一次连接异常恢复引擎 HttpEngine
```

```
retryEngine = engine.recover(e.getLastConnectException(),
null); // 如果恢复成功, 将当前的引擎设置为这个恢复好的引擎
if (retryEngine != null) { releaseConnection =
false; engine = retryEngine; continue; }
// 没法恢复, 抛出异常 throw e.getLastConnectException();
} catch (IOException e) { // 与服务器交互失败, 这时, 请求可
能已经被发送 // 恢复引擎 HttpEngine retryEngine =
engine.recover(e, null); //如果恢复成功, 将当前的引擎设置为
这个恢复好的引擎 if (retryEngine != null) {
releaseConnection = false; engine = retryEngine;
continue; } // 没法恢复, 抛出异常 throw e; }
finally { // 如果需要释放连接, 则将连接释放 if
(releaseConnection) { StreamAllocation
streamAllocation = engine.close();
streamAllocation.release(); } } // 获取响应 Response
response = engine.getResponse(); // 下一步的请求, 如果存在,
则需要重定向 Request followUp = engine.followUpRequest();
//如果不需要重定向 if (followUp == null) { if
(!forWebSocket) { // 释放连接
engine.releaseStreamAllocation(); } //返回响应
return response; } // 如果需要重定向, 关闭当前引擎
StreamAllocation streamAllocation = engine.close(); // 如
果超过最大数, 释放连接, 并抛出异常 if (++followUpCount >
MAX_FOLLOW_UPS) { // 释放连接
streamAllocation.release(); // 抛出异常 throw new
ProtocolException("Too many follow-up requests: " +
followUpCount); } // 如果重定向的地址和当前的地址一样, 则不需
要释放连接 if (!engine.sameConnection(followUp.url())) {
streamAllocation.release(); streamAllocation = null;
} // 使用重定向后的请求, 重新实例一个引擎, 开始下一次循环
request = followUp; engine = new HttpEngine(client,
request, false, false, forWebSocket, streamAllocation,
null, response); } } }
```

## 总结

RealCall的作用一句话概括——发送请求。其中细节有拦截器的建立过  
程, 异步回调的调用, HttpEngine的使用过程。

HttpEngine的代码没有往里Step in, 本系列以后的文章会有分析。

2.根据所看的文章，及源码的查看，自己做了简略的总结

Okhttp3：从以下方面总结

## 为什么okHttp3 好用呢？

OkHttp是一个精巧的网络请求库,有如下特性:

- 1)支持http2，对一台机器的所有请求共享同一个socket
- 2)内置连接池，支持连接复用，减少延迟
- 3)支持透明的gzip压缩响应体
- 4)通过缓存避免重复的请求
- 5)请求失败时自动重试主机的其他ip，自动重定向
- 6)好用的API

## 实现网络请求方法:

OkHttp3的最底层是Socket，而不是URLConnection，它通过Platform的Class.forName()反射获得当前Runtime使用的socket库

socket发起网络请求的流程一般是：

- (1). 创建socket对象;
- (2). 连接到目标网络;
- (3). 进行输入输出流操作。

(1)(2)的实现，封装在connection接口中，具体的实现类是RealConnection。

(3)是通过stream接口来实现，根据不同的网络协议，有Http1xStream和Http2xStream两个实现类

由于创建网络连接的时间较久(如果是HTTP的话，需要进行三次握手)，而请求经常是频繁的碎片化的，所以为了提高网络连接的效率，OKHttp3实现了网络连接复用

## 运用到的设计模式：

**单例模式**：（建议用单例模式创建OkHttpClient） OkHttpClient， 可以通过 new OkHttpClient() 或 new OkHttpClient.Builder() 来创建对象，但是---特别注意， OkHttpClient() 对象最好是共享的， 建议使用单例模式创建。因为每个 OkHttpClient 对象都管理自己独有的线程池和连接池。这一点很多同学，甚至在我经历的团队中就有人踩过坑，每一个请求都创建一个 OkHttpClient 导致内存爆掉

**外观模式**：OkHttpClient 里面组合了很多的类对象。其实是将OKHttp 的很多功能模块，全部包装进这个类中，让这个类单独提供对外的API，这种设计叫做外观模式（外观模式：隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口）

**Builder模式**：OkHttpClient 比较复杂，太多属性，而且客户的组合需求多样化，所以OKhttp 使用建造者模式（Build模式：使用多个简单的对象一步一步构建成一个复杂的对象，一个 Builder 类会一步一步构造最终的对象）

**工厂方法模式**：Call 接口提供了内部接口 Factory(用于将对象的创建延迟到该工厂类的子类中进行，从而实现动态的配置，工厂方法模式。（工厂方法模式：这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。）

**享元模式**：在 Dispatcher 的线程池中，所用到了享元模式，一个不限容量的线程池，线程空闲时存活时间为 60 秒。线程池实现了对象复用，降低线程创建开销，从设计模式上来讲，使用了享元模式。（享元模式：尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象，主要用于减少创建对象的数量，以减少内存占用和提高性能）

**责任链模式**：很明显，在okhttp中的拦截器模块，执行过程用到。OkHttp3 的拦截器链中，内置了5个默认的拦截器，分别用于重试、请求对象转换、缓存、链接、网络读写（责任链模式：为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解

耦。这种类型的设计模式属于行为型模式。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。）

**策略模式**：CacheInterceptor 实现了数据的选择策略，来自网络还是来自本地？这个场景也是比较契合策略模式场景，CacheInterceptor 需要一个策略提供者提供它一个策略（锦囊），CacheInterceptor 根据这个策略去选择走网络数据还是本地缓存。

缓存的策略过程：

- 1、 请求头包含 "If-Modified-Since" 或 "If-None-Match" 暂时不走缓存
- 2、 客户端通过 cacheControl 指定了无缓存，不走缓存
- 3、 客户端通过 cacheControl 指定了缓存，则看缓存过期时间，符合要求走缓存。
- 4、 如果走了网络请求，响应状态码为 304（只有客户端请求头包含 "If-Modified-Since" 或 "If-None-Match"，服务器数据没变化的话会返回 304 状态码，不会返回响应内容），表示客户端继续用缓存。

（策略模式：一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。）

源码中用到的几个重要的类及作用解释：

**1. OkHttpClient** :对外的API，OKHttp的很多功能模块，全部包装进这个类；创建分为两种：一种是new OkHttpClient()的方式，另一种是使用建造者（Builder）模式 -- new OkHttpClient.Builder()....Build()。那么这两种方式有什么区别呢？

第一种：new OkHttpClient()，OkHttp做了很多工作，很多我们需要的参数在这里都获得默认值,也就是默认值设定。

第二种：默认的设置和第一种方式相同，但是我们可以利用建造者模式单独的设置每一个属性；

注意事项：OkHttpClient强烈建议全局单例使用，因为每一个 OkHttpClient都有自己单独的连接池和线程池，复用连接池和线程池能够减少延迟、节省内存。

**2.RealCall类：**集成Call类，从源代码中，可看到使用Call类，发送出（同步 / 异步）请求.RealCall的主要作用：发送请求，当中还有拦截器的建立过程，异步回调。

**3.Dispatcher类（调度器，多线程）：**保存同步和异步Call的地方，并负责执行异步AsyncCall

**4.拦截器链：**有用户自定义的interceptor、  
retryAndFollowUpInterceptor、BridgeInterceptor、  
CacheInterceptor、ConnectInterceptor、networkInterceptors、  
CallServerInterceptor。依次通过以上拦截器，传递给RealCall中的  
ApplicationInterceptorChain。拦截器之所以可以依次调用，并最终再从  
后先前返回Response，都依赖于ApplicationInterceptorChain的  
proceed方法。

**5.HttpEngine类：**OKhttp底层的实现，（还在看）

缓存策略：提到缓存策略，就要提到CacheInterceptor拦截器，如下图



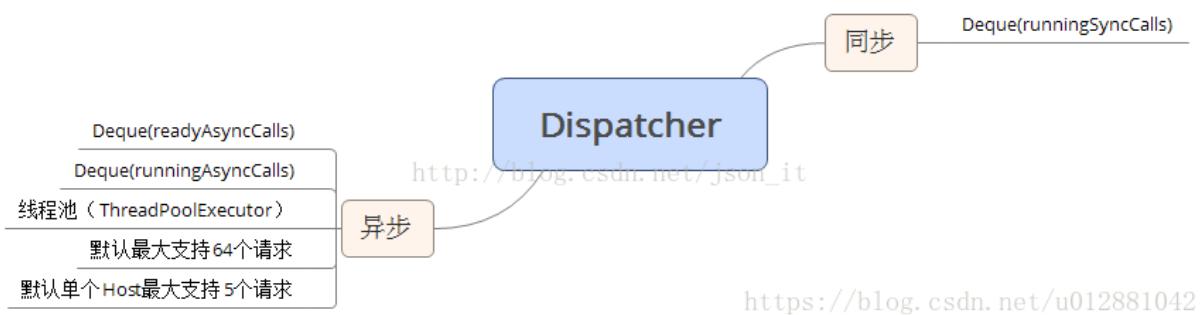
<https://blog.csdn.net/u012881042>

CacheStrategy实现缓存策略，CacheStrategy使用[Factory模式](#)进行构造，该类决定是使用缓存还是使用网络请求

Cache是封装了实际的缓存操作；

DiskLruCache: Cache 基于 DiskLruCache;

线程池（同步，异步）：



针对异步请求，Dispatcher使用了两个Deque，一个保存准备执行的请求，一个保存正在执行的请求，为什么要用两个呢？因为Dispatcher默认支持最大的并发请求是64个，单个Host最多执行5个并发请求，如果超过，则Call会先被放入到readyAsyncCall中，当出现空闲的线程时，再将readyAsyncCall中的线程移入到runningAsynCalls中，执行请求。

如果正在执行的请求总数 $\leq 64$  && 单个Host正在执行的请求 $\leq 5$ ，则将请求加入到runningAsyncCalls集合中，紧接着就是利用线程池执行该请求，否则就将该请求放入readyAsyncCalls集合中。

未完待续。

## 第九节 okhttp连接池复用机制

### 9.1 概述

| 提高网络性能优化，很重要的一点就是降低延迟和提升响应速度。

通常我们在浏览器中发起请求的时候header部分往往是这样的

▼ Response Headers [view source](#)

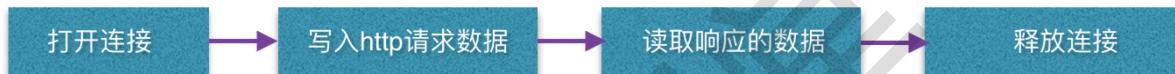
Cache-Control: private, max-age=10  
Connection: keep-alive keep-alive  
Content-Encoding: gzip  
Content-Type: text/html; charset=utf-8  
Date: Tue, 21 Jun 2016 08:28:22 GMT

`keep-alive` 就是浏览器和服务端之间保持长连接，这个连接是可以复用的。在HTTP1.1中是默认开启的。

连接的复用为什么会提高性能呢？

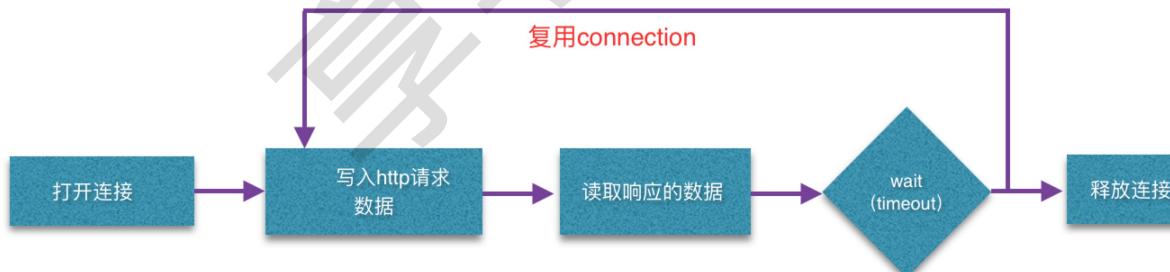
通常我们在发起http请求的时候首先要完成tcp的三次握手，然后传输数据，最后再释放连接。三次握手的过程可以参考[这里 TCP三次握手详解及释放连接过程](#)

一次响应的过程



在高并发的请求连接情况下或者同个客户端多次频繁的请求操作，无限制的创建会导致性能低下。

如果使用 `keep-alive`



在 `timeout` 空闲时间内，连接不会关闭，相同重复的request将复用原先的 `connection`，减少握手的次数，大幅提高效率。

并非 `keep-alive` 的 `timeout` 设置时间越长，就越能提升性能。长久不关闭会造成过多的僵尸连接和泄露连接出现。

那么okhttp在客户端是如果类似于客户端做到的 `keep-alive` 的机制。

## 9.2 连接池的使用

连接池的类位于`okhttp3.ConnectionPool`。我们的主旨是了解到如何在`timeout`时间内复用connection，并且有效的对其进行回收清理操作。

其成员变量代码片

```
/** * Background threads are used to cleanup expired connections. There will be at most a single * thread running per connection pool. The thread pool executor permits the pool itself to be * garbage collected. */
private static final Executor executor = new ThreadPoolExecutor(0 /* corePoolSize */,
Integer.MAX_VALUE /* maximumPoolsize */, 60L /* keepAliveTime */,
TimeUnit.SECONDS, new SynchronousQueue<Runnable>(),
util.threadFactory("okHttp ConnectionPool", true)); /** The maximum number of idle connections for each address. */
private final int maxIdleConnections;
private final Deque<RealConnection> connections = new ArrayDeque<>();
final RouteDatabase routeDatabase = new RouteDatabase();
boolean cleanupRunning;
```

- `excutor` : 线程池，用来检测闲置socket并对其进行清理。
- `connections` : connection缓存池。`Deque`是一个双端列表，支持在头尾插入元素，这里用作LIFO（后进先出）堆栈，多用于缓存数据。
- `routeDatabase` : 用来记录连接失败router

### 9.2.1 缓存操作

`ConnectionPool`提供对`Deque<RealConnection>`进行操作的方法分别为`put`、`get`、`connectionBecameIdle`、`evictAll`几个操作。分别对应放入连接、获取连接、移除连接、移除所有连接操作。

`put`操作

```
void put(RealConnection connection) { assert
(Thread.holdsLock(this)); if (!cleanupRunning) {
 cleanupRunning = true;
 executor.execute(cleanupRunnable); }
 connections.add(connection); }
```

可以看到在新的connection放进列表之前执行清理闲置连接的线程。

既然是复用，那么看下他获取连接的方式。

```
/** Returns a recycled connection to {@code address}, or
null if no such connection exists. */RealConnection
get(Address address, StreamAllocation streamAllocation) {
 assert (Thread.holdsLock(this)); for
(RealConnection connection : connections) { if
(connection.allocations.size() <
connection.allocationLimit &&
address.equals(connection.route().address) &&
!connection.noNewStreams) {
 streamAllocation.acquire(connection); return
connection; } } return null; }
```

遍历connections缓存列表，当某个连接计数的次数小于限制的大小以及request的地址和缓存列表中此连接的地址完全匹配。则直接复用缓存列表中的connection作为request的连接。

streamAllocation.allocations是个对象计数器，其本质是一个  
`List<Reference<StreamAllocation>>` 存放在RealConnection连接  
对象中用于记录Connection的活跃情况。

连接池中Connection的缓存比较简单，就是利用一个双端列表，配合CRD等操作。那么connection在timeout时间类是如果失效的呢，并且如果做到有效的对连接进行清除操作以确保性能和内存空间的充足。

## 9.2.2 连接池的清理和回收

在看ConnectionPool的成员变量的时候我们了解到一个Executor的线程池是用来清理闲置的连接的。注释中是这么解释的：

Background threads are used to cleanup expired connections

我们在put新连接到队列的时候会先执行清理闲置连接的线程。调用的正是 `executor.execute(cleanupRunnable);` 方法。观察 `cleanupRunnable`

```
private final Runnable cleanupRunnable = new Runnable() {
 @Override public void run() { while (true) {
 long waitNanos = cleanup(System.nanoTime()); if
(waitNanos == -1) return; if (waitNanos > 0) {
 long waitMillis = waitNanos / 1000000L;
 waitNanos -= (waitMillis * 1000000L);
 synchronized (ConnectionPool.this) { try {
 ConnectionPool.this.wait(waitMillis, (int)
waitNanos); } catch (InterruptedException
ignored) { } } } } } };
```

线程中不停调用 `cleanup` 清理的动作并立即返回下次清理的间隔时间。继而进入 `wait` 等待之后释放锁，继续执行下一次的清理。**所以可能理解成他是个监测时间并释放连接的后台线程。**

了解 `cleanup` 动作的过程。这里就是如何清理所谓闲置连接的和行了。怎么找到闲置的连接是主要解决的问题。

```
long cleanup(long now) { int inUseConnectionCount = 0;
 int idleConnectionCount = 0; RealConnection
longestIdleConnection = null; long
longestIdleDurationNs = Long.MIN_VALUE; // Find either
a connection to evict, or the time that the next eviction
is due. synchronized (this) { for
(Iterator<RealConnection> i = connections.iterator();
i.hasNext();) { RealConnection connection =
i.next(); // If the connection is in use, keep
searching. if
(pruneAndGetAllocationCount(connection, now) > 0) {
 inUseConnectionCount++; continue; }
 idleConnectionCount++; // If the connection
is ready to be evicted, we're done. long
idleDurationNs = now - connection.idleAtNanos; if
(idleDurationNs > longestIdleDurationNs) {
 longestIdleDurationNs = idleDurationNs;
 longestIdleConnection = connection; }
 } if (longestIdleDurationNs >= this.keepAliveDurationNs
|| idleConnectionCount > this.maxIdleConnections) {
 // We've found a connection to evict. Remove it
from the list, then close it below (outside // of
the synchronized block).
connections.remove(longestIdleConnection); } else if
(idleConnectionCount > 0) { // A connection will
be ready to evict soon. return keepAliveDurationNs
- longestIdleDurationNs; } else if
(inUseConnectionCount > 0) { // All connections
are in use. It'll be at least the keep alive duration
'til we run again. return keepAliveDurationNs;
} else { // No connections, idle or in use.
cleanupRunning = false; return -1; } }
closeQuietly(longestIdleConnection.socket()); // Clean up again immediately.
return 0; }
```

在遍历缓存列表的过程中，使用连接数目 `inUseConnectionCount` 和闲置连接数目 `idleConnectionCount` 的计数累加值都是通过 `pruneAndGetAllocationCount()` 是否大于0来控制的。那么很显然 `pruneAndGetAllocationCount()` 方法就是用来识别对应连接是否闲置的。>0则不闲置。否则就是闲置的连接。

进去观察

```
private int pruneAndGetAllocationCount(RealConnection connection, long now) {
 List<Reference<StreamAllocation>> references =
 connection.allocations; for (int i = 0; i <
 references.size();) { Reference<StreamAllocation>
 reference = references.get(i); if (reference.get()
 != null) { i++; continue; } //
 We've discovered a leaked allocation. This is an
 application bug. Platform.get().log(WARN, "A
 connection to " + connection.route().address().url()
 + " was leaked. Did you forget to close a response
 body?", null); references.remove(i);
 connection.noNewStreams = true; // If this was the
 last allocation, the connection is eligible for immediate
 eviction. if (references.isEmpty()) {
 connection.idleAtNanos = now - keepAliveDurationNs;
 return 0; } } return references.size(); }}
```

好了，原先存放在 `RealConnection` 中的 `allocations` 派上用场了。遍历 `StreamAllocation` 弱引用链表，移除为空的引用，遍历结束后返回链表中弱引用的数量。所以可以看出

`List<Reference<StreamAllocation>>` 就是一个记录 `connection` 活跃情况的 >0 表示活跃 =0 表示空闲。`StreamAllocation` 在列表中的数量就是物理socket被引用的次数

解释：`StreamAllocation` 被高层反复执行 `aquire` 与 `release`。这两个函数在执行过程中其实是在一直在改变 `Connection` 中的 `List<WeakReference<StreamAllocation>>` 大小。

搞定了查找闲置的 connection 操作，我们回到 cleanup 的操作。计算了 inUseConnectionCount 和 idleConnectionCount 之后程序又根据闲置时间对 connection 进行了一个选择排序，选择排序的核心是：

```
// If the connection is ready to be evicted, we're
done. long idleDurationNs = now -
connection.idleAtNanos; if (idleDurationNs >
longestIdleDurationNs) { longestIdleDurationNs =
idleDurationNs; longestIdleConnection =
connection; } ...
```

通过对最大闲置时间选择排序可以方便的查找出闲置时间最长的一个 connection。如此一来我们就可以移除这个没用的 connection 了！

```
if (longestIdleDurationNs >= this.keepAliveDurationNs
|| idleConnectionCount > this.maxIdleConnections)
{ // We've found a connection to evict. Remove it
from the list, then close it below (outside // of
the synchronized block).
connections.remove(longestIdleConnection);}
```

总结：清理闲置连接的核心主要是引用计数器

List<Reference<StreamAllocation>> 和 选择排序的算法 以及  
executor 的清理线程池。

## 第十节 okhttp 流程和优化的实现

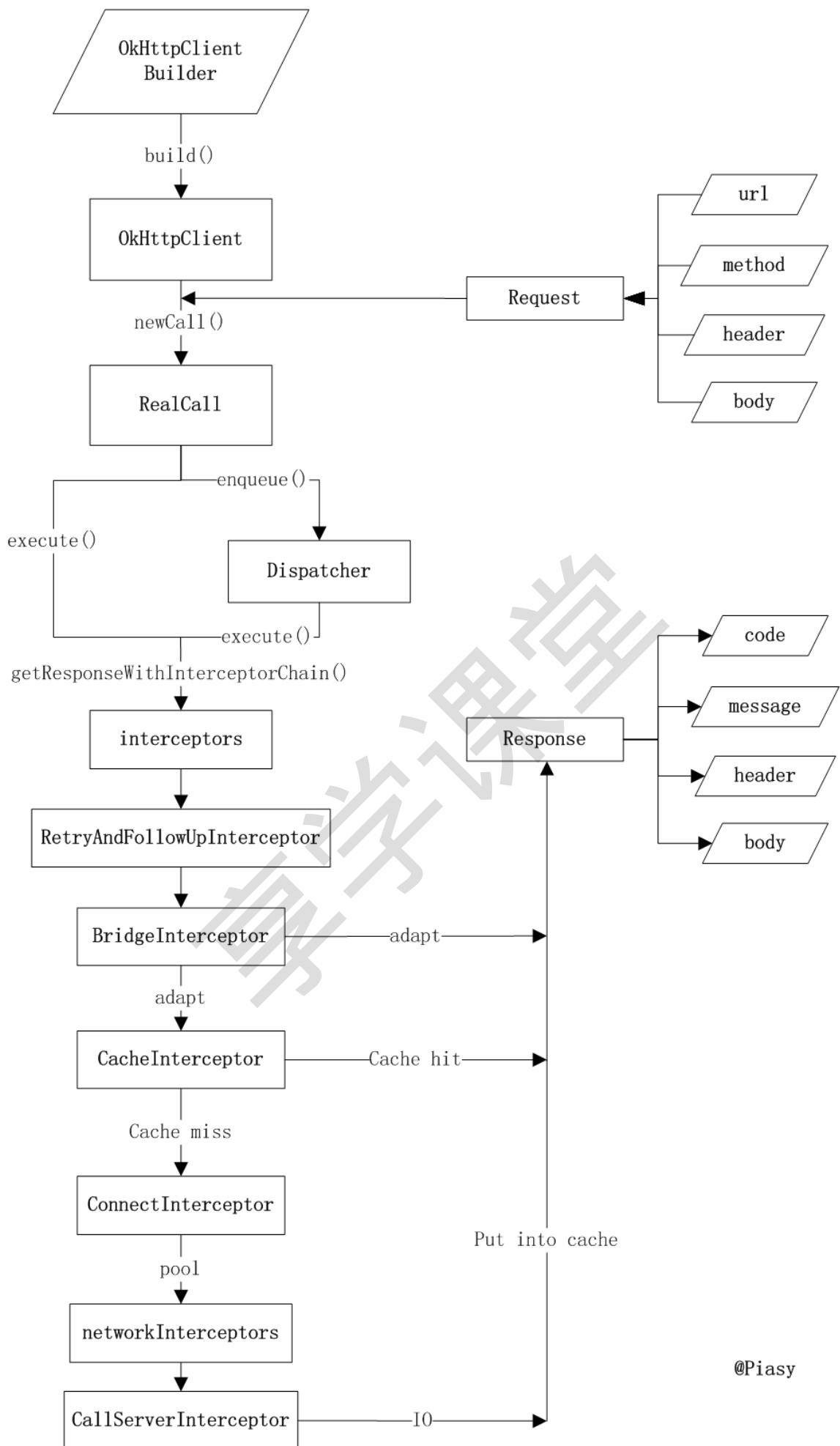
### 10.1 概述

最近一直在忙着研究 okhttp，看了两周了，感觉东西实在是太多了，如果让我细致的写，我感觉能写 10 篇都写不完，那东西虽然是很多，但是主要的流程我们还是需要了解的，这篇文章我主要介绍以下 okhttp 中的流程，还有一些做的好的东西，之后再去将一些细节上的东西，这篇文章主要说了一个大概的内容，就是帮助你大致了解一下 okhttp 的实现，废话不多说了下面进入正题。

## 10.2 异步流程的实现

我们首先看一下网络上面很流行的一张图片，这张图片基本涵盖了一个整个的流程，下面我就对这张图片进行一一的说明：





@Piassy

1、当我们请求的时候会新建一个 RealCall 的对象，创建之后会通过 dispatcher 去在线程池中分配线程，这个 dispatcher 的主要作用就是调度请求，这里面有三个队列，作用分别是存储异步正在运行的任务，存储异步正在准备运行的任务，还有同步运行的队列，通过这个类为我们的任务分配一个线程去运行

2、获取 Response 的过程，我们首先通过递归（index+1 的方式实现递归的）的方式一个一个的加入拦截器，拦截器首先加入的是我们的自定义的拦截器，然后去加框架自己的拦截器，那这几个拦截器的主要功能如下：

1、RetryAndFollowUpInterceptor (负责失败重试，重定向的)

概述：主要的作用就是请求时候创建一个 StreamAllocation 对象，这个对象创建的时候会根据请求的协议不同创建不一样的对象

2、BridgeInterceptor(负责把用户构造的请求转换为发送到服务器的请求、把服务器返回的响应转换为用户友好的响应的)

概述：主要就是把我们的 request 请求加上一些请求头，打包成真正的网络请求的 request，在请求返回的时候，通过 gzip 把我们能的 response 进行压缩

3、CacheInterceptor(负责读取缓存的，如果有缓存就拦截并返回，也负责更新缓存)

概述：负责读取我们的缓存，如果我们设置了 cache，那么先从这个里面读取，如果读取不到的那么就把 request 和 caseResponse 构建一个 CacheStrategy 对象，然后判断这个对象是否有效，如果有效则直接返回，如果无效，那么我们请求，如果请求返回 304，证明资源没过期，我们可以读取本地的缓存

4、ConnectInterceptor(负责和服务器建立连接的)

概述：这个就比较复杂了，这个里面主要是进行 socket 的连接，在这个拦截器里首先获取了一个 streamAllocation 对象，然后通过这个对象获取了一个 RealConnection 对象，然后通过这个对象去获取一个 httpcode 对象，这个对象是一个接口，那具体实现有两种一个是 http1，一个是 http2，它会根据我们的请求创建不同的 httpCode，通过这个对象可以进行下一个拦截器的操作，在我们获取 realconnection 的同时我们调用了 connect 这个方法，这个方法底层调用的就是 socket.connect 的方法，就是进行了 socket 的连接了

5、配置 OkHttpClient 时设置的 networkInterceptors

6、CallServerInterceptor(负责向服务器发送请求数据、从服务器读取响应数据的)

概述：通过4创建的 httpcode 进行写入数据，它是通过 okio 的 sink 进行写入数据的，这个 sink 和outputstream 差不多，就相当于写入流中数据，然后发送到服务器，这是请求部分。处理 Response 的时候，他会读取我们的返回的数据，通过 Source，然后把数据封装成 response 对象

注意：我们每个拦截器在 intercept 方法中都会调用 process 这个方法，这个方法的作用就是去执行下一个拦截器，那么如果我们自定义拦截器的话，需要调用 chain.proceed 方法，不然的话我们的请求就会在拦截器里面卡住。

那么异步请求的东西大致就这么多了，还有很多细致的东西之后会分享出来。

## 10.3 okhttp 中有哪些优化，优化是怎么实现的

1、多路复用

概述：之前我们的请求是每一次请求会建立一个链接，请求结束就关闭链接，我们都知道 TCP/IP 请求是需要握手的，那握手就会消耗相应的时间，所以在我们的 okhttp 中，我们会复用之前的链接进行请求，这样请求速度就快了很多。

如果是这样的话就会出现两个问题，第一，我们怎么判断链接是否可用，第二是我们不需要的链接怎么回收。

从上面我们知道我们会在 StreamAllocation.newStream 方法中获取 RealConnection，在获取的时候我们会判断有没有之前的链接可以复用，复用的条件是这样判断的：

- 1、如果此链接的负载数目超过指定数目（表现为RealConnection的 allocations集合的数量超过该链接指定的数量）或者noNewStreams为 true 时，此链接不可复用。
- 2、StreamAllocation 所持有的Address对象和RealConnection的 Address非主机部分不同，则此链接不可复用。至于非主机部分的判定是在Address的equalsNonHost方法来体现。

两者Adress对象的非主机部分相等的标准就是dns,Authenticator对象、协议、CA授权验证标准、端口等信息全部相等。

- 3、在1、2判定条件都为true的话，如果两个Address对象的host或者说 url中的host一样，则此链接可复用，正如注释说说，添加1、2、3都满足的话，那么此时这个链接就是This connection is a perfect match。

第一个问题我们解决了，现在我们来解决第二个问题，我们都知道链接如果多了我们如果不回收的话就会卡死，那么我们的链接是怎么回收的呢，链接回收主要降到的就是 ConnectionPool 这个类，这个类中有一个 clean up 方法，我们来看一下他里面做了什么：

clean up 方法：

标记清除法

- 1、首先标记出最不活跃的链接（空闲链接），之后进行清除
- 2、如果被标记的链接空闲 socket 超过 5 个，时间大于 5 分钟，那么直接清除。

3、如果此链接空闲，但是不足五分钟，则返回剩余时间，并进行标记，以供下次清除。

4、如果没用空闲链接的话，则五分钟之后再进行清理

判断是否为空闲链接：

1、遍历其中的 StreamAllocation，判断是否为空，如果为空，则没有引用这个 StreamAllocation

2、如果引用数量为 0，则为空闲链接

多路复用的原理就讲到这里了，其实很简单，只要吧 ConnectionPool 这个类看明白就很好理解了

## 2、缓存

概述：我们都知道，好的框架都会有缓存的功能，通过缓存我们可以很快的访问我们的资源，那 okhttp 也不例外，从上面的流程中我们可以看到， CacheInterceptor 主要是做缓存的，那么我们来了解一下他的流程是什么：

okhttp 的缓存策略是，key 为 Request 的 url 的 MD5 值，value 为 response。

1、如果在 okhttpclient 初始化的时候配置了 cache，那么我们则从缓存中读取 caseResponse。

2、如果没有指定，那么我们将 request 和 caseResponse 构建一个 CacheStrategy 的类

3、判断 cachestrategy 是否有效，如果 request 和 caseResponse 都为空，直接返回 504

4、如果 request == null，cacheResponse 不为空，则返回

5、如果为空，那么我们就进行网络请求，如果返回了 304 且我们本地有缓存，那么说明我们的缓存没有过期，可以继续使用

### 3、线程池的引入

概述：我们之前的请求都是每次都会新建线程去进行请求，这样的话我们如果有 100 个请求就会有 100 个线程，那么会消耗很大的资源，当引入线程池之后，我们就不需要频繁的去创建线程，而且可以复用线程，这样就很节省时间了。

### 4、可以进行压缩

在流程中我们讲到，当我们 response 通过 bridgeInterceptor 处理的时候会进行 gzip 压缩，这样可以大大减小我们的 response，他不是什么情况下都压缩的，只有支持的时候才会进行压缩。

其实还有一个是重连和重定向，现在还没弄懂，之后再去分享吧

## 10.4 okhttp 中用到了哪些设计模式

1、建造者模式，不管是构建一个 OkHttpClient 或者 Request 都是很复杂的，所以在 okhttp 框架中使用了建造者模式去创建这两个对象。

2、其实我认为 okhttp 最好的就是他的拦截器的设计，通过递归调用的方法去一个一个的拦截，最后在一个一个的处理，这样形成一条链路，让我们操作起来非常方便

总结：

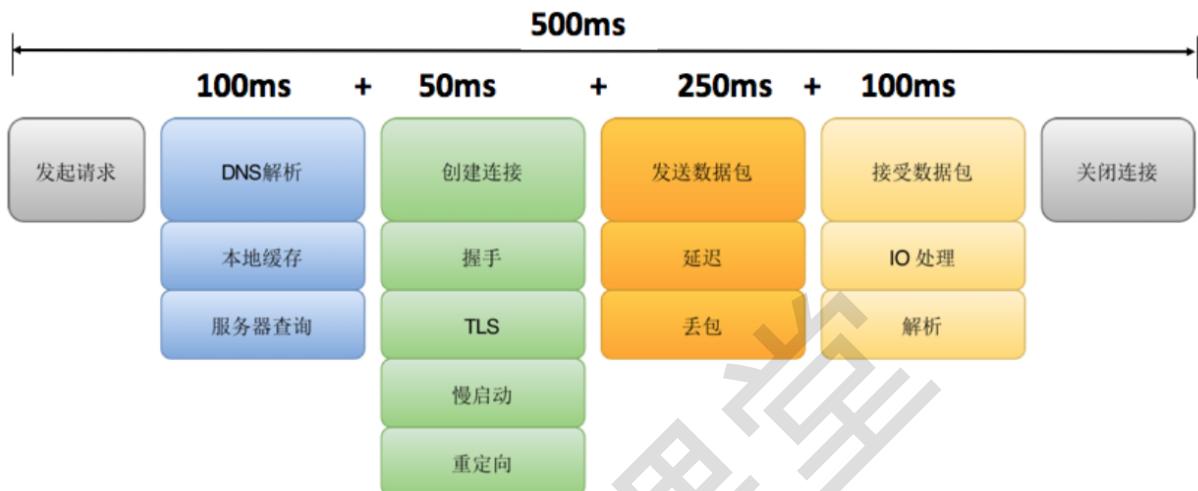
好了，今天的东西就讲到这里吧，其实主要讲的都是一些流程上面的东西，而且讲的很粗略，没有那么细致，如果想深入了解的话还需要去看一下源码是怎么写的，这样会很深入的理解 okhttp。

## 第十一节 一篇让你受用的okhttp分析

本文希望你在读完之后能够清楚的事情：一次http请求的经历，这期间会遇到什么问题，okhttp怎么解决的，在它的责任链中的那一部分解决的这个问题，怎样监控我们自己的网络请求，怎样监控网络状况。

## 一次http网络请求的历程

网络请求要依次经历 DNS解析、创建连接、收发数据、关闭连接几个过程。下图是其他教程里的一张图，画的非常清晰：



网络请求的过程.png

这期间需要应对的问题有：

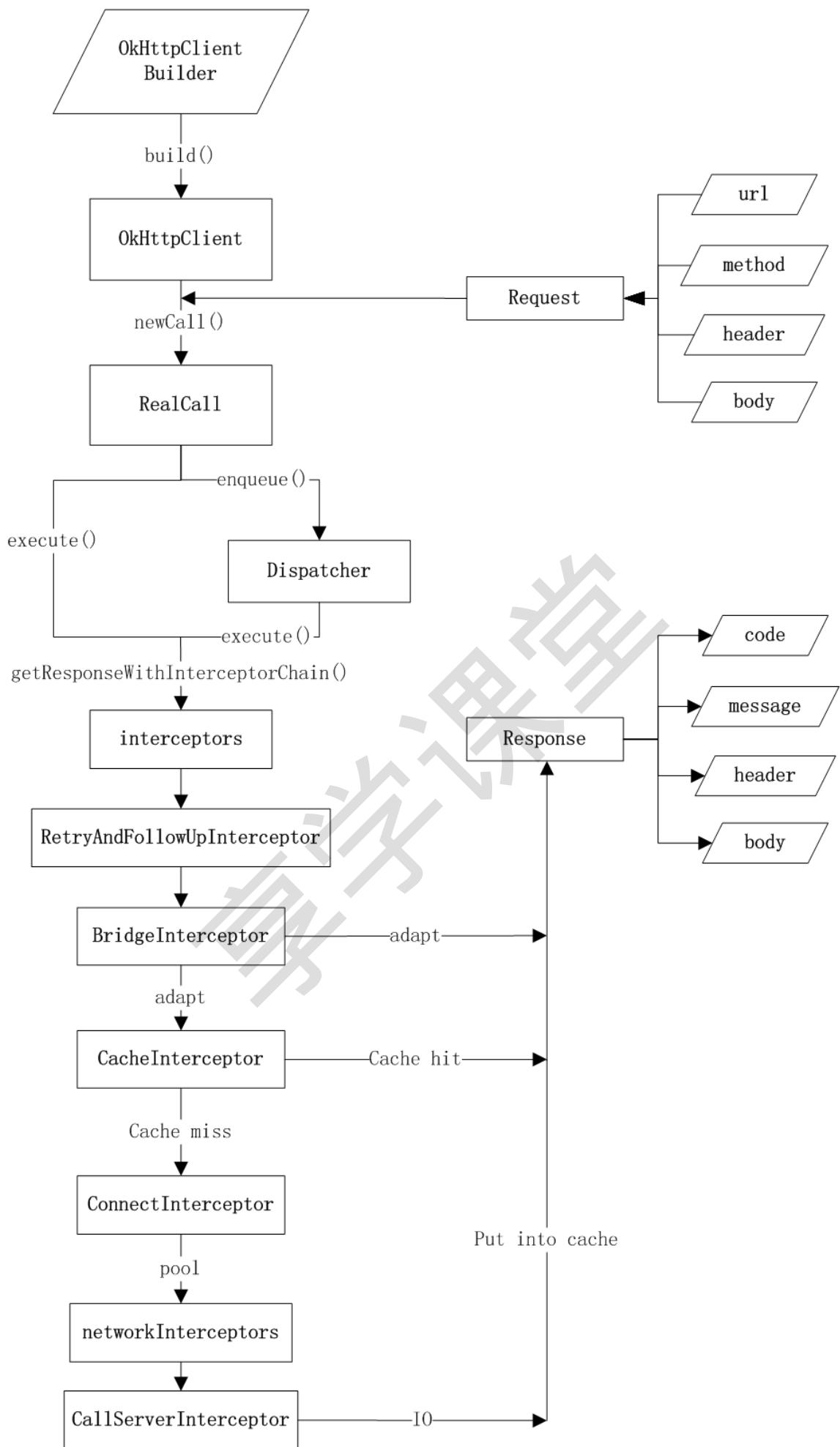
- 1、DNS劫持。即使我们现在几万用户的小体量app，每月也能碰到几起DNS劫持。除了DNS劫持，这部分还需要考虑IP选取策略、DNS缓存、容灾等问题，如果必要的话，可以对其进行优化，参考百度的[DNS优化](#)，以及[美图案例](#)(考虑了非okhttp的情况)。
- 2、连接复用。http基于TCP，所以连接要经历三次握手，关闭连接要经历4次握手，TCP连接在发送数据的时候，在起初会限制连接速度，随着传输的成功和时间的推移逐渐提高速度（防拥塞），再加上TLS密匙协商，如果每次网络请求都要经历创建连接的过程，带来的开销是非常大的。
- 3、I/O问题。客户端会等待服务器的返回数据，数据收到后还要把数据从内核copy到用户空间，期间根据网络的阻塞模型（基本有五种，常见的有阻塞I/O、[非阻塞I/O](#)、[多路复用I/O](#)），会遇到不同程度的阻塞。
- 4、数据压缩和加密。减少数据体积，对数据进行加密。

对于上述问题的方案：

- 1、okhttp提供了自定义DNS解析的接口。
- 2、持久连接。http1.1支持事务结束之后将TCP保持在打开状态 (http1.1默认将Keep-Alive首部开启，用于客户端和服务器通信连接的保存时间，TCP中有Keep-Alive报文，来定时探测通信双方是否存活，但是这一部分内容用于长连接时会存在问题)，对http1.1进行连接复用，将连接放入连接池。支持http2.0，http2.0使用多路复用，支持一条连接同时处理多个请求，请求可以并发进行，一个域名会保留一条连接 (一条连接即一个TCP连接，收发只有一根管道，并不是真正意义上的并发，而是利用TCP把数据拆分装包加标签的特性实现的复用)，能有效降低延时 (也有特殊情况，比如一个域名的数据请求特别多，或者服务端对单个连接有速度限制，如视频流)。[长连接](#) (推荐读下这篇文章，如果没有从事过长连接开发的话) 也是一种方案，在某些场景下非常有效，但是okhttp不支持。
- 3、OKhttp使用非阻塞I/O模型OKio(算是nio吧，和我们理解的nio不太一致，理解的nio定时去检查是否有数据到来，有的话就读，没有就返回，但是okio的实现是定时去检查是否已经读写完成，没完成就认为超时，close掉该socket)，该I/O框架的内存表现也很好 (mars使用epoll)。
- 4、http2.0协议本身对头部有压缩。对于body的压缩okhttp提供了Gzip压缩的支持。

## OKhttp实现分析

网上找到的一个okhttp整体调用图，由于okhttp的分析已经很多，尽量以少代码多总结的方式来阐述这部分内容。主要讲解okhttp应对上述问题的具体实现。



okhttp整体调用图.png

设计结构十分清晰，通过责任链将请求发送任务进行拆解。

## 1、okhttp中自定义DNS解析

```
OkHttpClient client = new OkHttpClient.Builder()
 .dns(new Dns() {
 @Override
 public List<InetAddress> lookup(String hostname) throws
 UnknownHostException {
 return
 Arrays.asList(InetAddress.getAllByName(DNSHelper.getIpByH
 ost(hostname)));
 }
 })
 .build();
```

DNSHelper可以通过ip直连的方式访问自己设置的DNS服务器。也可以通过这种方式接入一些第三方对外提供的DNS服务。

在RetryAndFollowUpInterceptor这个拦截器中，将会创建Address对象，该对象收集网络请求需要的配置信息，包括DNS、host、port、proxy等。在ConnectInterceptor这一层创建了RouteSelecter，用于路由选择，持有Address对象，调用其next函数选择路由时会调用DNS对象的lookup函数返回host的ip。

## 2、连接复用

这部分需要操心的事情一个是连接的管理，一个是对网络请求的流的管理。连接的管理交由ConnectionPool，内部含有一个保存了RealConnection对象的队列。http2.0一个连接对应多个流，在RealConnection内保存了一个代表流的StreamAllocation对象的list。

在ConnectInterceptor这一层调用StreamAllocation的新Strem，尝试在连接池里找到一个RealConnection，没找到则创建一个，并调用acquire添加一个自身的弱引用到RealConnection的流引用List中。

newStrem最终返回一个httpcodec接口的实现，代表了具体的http协议内容创建规则，有两种实现，对应了okhttp适配的两个http版本，然后传递给下一级。当然流在读写完成后也是需要被清理的，清理函数deallocate，一个连接的流都被清理掉之后，通知ConnectionPool判断连接的keep-alive时间，以及空闲连接数量，移除超时或者超出数量限制后空闲时间最长的连接。

如下是调用流和连接的绑定过程（省略了路由选择过程），新创建的连接会执行socket的connect，connect的时候会判断http协议是哪个版本，然后新创建的RealConnection会添加到连接池里。

```
// Attempt to use an already-allocated connection.
RealConnection allocatedConnection = this.connection; if
(allocatedConnection != null &&
!allocatedConnection.noNewStreams) { return
allocatedConnection; } // Attempt to get a connection
from the pool. Internal.instance.get(connectionPool,
address, this, null); if (connection != null) {
return connection; }RealConnection result; synchronized
(connectionPool) { if (canceled) throw new
IOException("Canceled"); // Create a connection and
assign it to this allocation immediately. This makes it
possible // for an asynchronous cancel() to interrupt
the handshake we're about to do. result = new
RealConnection(connectionPool, selectedRoute);
acquire(result);}// Do TCP + TLS handshakes. This is a
blocking operation.result.connect(connectTimeout,
readTimeout, writeTimeout,
connectionRetryEnabled);routeDatabase().connected(result.
route());Socket socket = null; synchronized
(connectionPool) { // Pool the connection.
Internal.instance.put(connectionPool, result); // If
another multiplexed connection to the same address was
created concurrently, then // release this connection
and acquire that one. if (result.isMultiplexed()) {
socket = Internal.instance.deduplicate(connectionPool,
address, this); result = connection;
}}closeQuietly(socket);return result;
```

如下可以看出连接池对空闲时间和空闲连接数量的限制（顺带一提，okhttp的线程池也是有数量限制的，大约在60个左右，如果项目网络库比较乱，使用线程也不太注意，线程过多，超过500个，在一些华为手机上会因为申请不到线程而崩溃）。

```
private final Deque<RealConnection> connections = new
ArrayDeque<>(); final RouteDatabase routeDatabase = new
RouteDatabase(); boolean cleanupRunning; /** * Create
a new connection pool with tuning parameters appropriate
for a single-user application. * The tuning parameters
in this pool are subject to change in future okhttp
releases. Currently * this pool holds up to 5 idle
connections which will be evicted after 5 minutes of
inactivity. */ public ConnectionPool() { this(5, 5,
TimeUnit.MINUTES); } public ConnectionPool(int
maxIdleConnections, long keepAliveDuration, TimeUnit
timeUnit) { this.maxIdleConnections =
maxIdleConnections; this.keepAliveDurationNs =
timeUnit.toNanos(keepAliveDuration); // Put a floor on
the keep alive duration, otherwise cleanup will spin
loop. if (keepAliveDuration <= 0) { throw new
IllegalArgumentException("keepAliveDuration <= 0: " +
keepAliveDuration); } }
```

如下是具体的清理逻辑：

```

 synchronized (this) { for
(Iterator<RealConnection> i = connections.iterator();
i.hasNext();) { RealConnection connection =
i.next(); // If the connection is in use, keep
searching. if
(pruneAndGetAllocationCount(connection, now) > 0) {
 inUseConnectionCount++; continue; }
 idleConnectionCount++; // If the connection
is ready to be evicted, we're done. long
idleDurationNs = now - connection.idleAtNanos; if
(idleDurationNs > longestIdleDurationNs) {
 longestIdleDurationNs = idleDurationNs;
 longestIdleConnection = connection; }
 if (longestIdleDurationNs >= this.keepAliveDurationNs
 || idleConnectionCount > this.maxIdleConnections) {
 // We've found a connection to evict. Remove it
from the list, then close it below (outside // of
the synchronized block). connections.remove(longestIdleConnection); } else if
(idleConnectionCount > 0) { // A connection will
be ready to evict soon. return keepAliveDurationNs
- longestIdleDurationNs; } else if
(inUseConnectionCount > 0) { // All connections
are in use. It'll be at least the keep alive duration
'til we run again. return keepAliveDurationNs;
} else { // No connections, idle or in use.
cleanupRunning = false; return -1; }
}

```

### 3、I/O优化

#### a、okio的使用：

```

okio.buffer(Okio.sink(socket)) .writeUtf8("write string
by utf-8.\n") .writeInt(1234).close();

```

b、okio没有使用java提供的select（多路复用），而是自定义了nio实现。个人猜测这样实现的原因是多路复用实际在网络连接非常多的时候表现更好，对于客户端来讲不一定适用，反倒会增加大量的select epoll系统调用，更多用于服务器。

设置一个watchdog，将一次事件（读、写）封装到AsyncTimeout中，  
AsyncTimeout持有一个static链表，Watchdog定期检测链表。

```
private static final class Watchdog extends Thread {
 Watchdog() { super("Okio Watchdog");
 setDaemon(true); } public void run() { while
 (true) { try { AsyncTimeout timedOut;
 synchronized (AsyncTimeout.class) {
 timedOut = awaitTimeout(); // Didn't find a
 node to interrupt. Try again. if (timedOut ==
 null) continue; // The queue is completely
 empty. Let this thread exit and let another watchdog
 thread // get created on the next call to
 scheduleTimeout(). if (timedOut == head) {
 head = null; return; }
 } // Close the timed out node.
 timedOut.timedOut(); } catch (InterruptedException
 ignored) { } } } }
```

awaitTimeout()函数读取链表，设置等待事件。到时间后，返回链表中的一个AsyncTimeout 对象，并调用该对象的timedOut()函数。

```

 static @Nullable AsyncTimeout awaitTimeout() throws
InterruptedException { // Get the next eligible node.
 AsyncTimeout node = head.next; // The queue is
empty. Wait until either something is enqueued or the
idle timeout elapses. if (node == null) {
 long
startNanos = System.nanoTime();
AsyncTimeout.class.wait(IDLE_TIMEOUT_MILLIS); return
head.next == null && (System.nanoTime() - startNanos) >=
IDLE_TIMEOUT_NANOS ? head // The idle timeout
elapsed. : null; // The situation has changed.
} long waitNanos =
node.remainingNanos(System.nanoTime()); // The head of
the queue hasn't timed out yet. Await that. if
(waitNanos > 0) { // Waiting is made complicated by
the fact that we work in nanoseconds, // but the API
wants (millis, nanos) in two arguments. long
waitMillis = waitNanos / 1000000L; waitNanos -=
(waitMillis * 1000000L);
AsyncTimeout.class.wait(waitMillis, (int) waitNanos);
return null; } // The head of the queue has timed
out. Remove it. head.next = node.next; node.next =
null; return node; }

```

而若这个事件已经完成则会调用exit()函数,将该事件在队列中移除。

```

/** Returns true if the timeout occurred. */ public
final boolean exit() { if (!inQueue) return false;
inQueue = false; return cancelScheduledTimeout(this);
} /** Returns true if the timeout occurred. */ private
static synchronized boolean
cancelScheduledTimeout(AsyncTimeout node) { // Remove
the node from the linked list. for (AsyncTimeout prev
= head; prev != null; prev = prev.next) { if
(prev.next == node) { prev.next = node.next;
node.next = null; return false; } } // The node wasn't
found in the linked list: it must have
timed out! return true; }

```

超时后认为连接不可用，调用Socket对象的close函数关闭该连接。

### c、缓存

是okio对java I/O 做的最重要的优化。主要思想是buffer复用，而不是创建大量的朝生夕死的buffer对象，防止频繁GC。这部分内容可以对比 BufferedInputStream的实现（BufferedInputStream内部结构和 Segment类似，当其设置的初始缓存byte数组大小不够时，新申请一个更大容量的数组，并将原缓存数组的内容copy过来，舍弃原数组）。

代码思路：Segment对象为byte数组的封装，是数据的容器，是一个双向链表中的节点，可以有插入、删除、拆分、合并、复制几个操作。

SegmentPool缓存了不用的segment，是一个静态的单链表，需要时调用take获取Segment，不需要时调用recycle回收。Buffer对象封装了这两者的使用，例如使用okio调用writeString函数时的实现如下：

```
 @Override public Buffer writeString(String string, int
beginIndex, int endIndex, Charset charset) { if
(string == null) throw new
IllegalArgumentException("string == null"); if
(beginIndex < 0) throw new IllegalAccessError("beginIndex
< 0: " + beginIndex); if (endIndex < beginIndex) {
throw new IllegalArgumentException("endIndex <
beginIndex: " + endIndex + " < " + beginIndex); }
if (endIndex > string.length()) { throw new
IllegalArgumentException("endIndex >
string.length: " + endIndex + " > " + string.length());}
 if (charset == null) throw new
IllegalArgumentException("charset == null"); if
(charset.equals(Util.UTF_8)) return writeUtf8(string,
beginIndex, endIndex); byte[] data =
string.substring(beginIndex, endIndex).getBytes(charset);
 return write(data, 0, data.length); } @Override
public Buffer write(byte[] source, int offset, int
byteCount) { if (source == null) throw new
IllegalArgumentException("source == null");
checkOffsetAndCount(source.length, offset, byteCount);
int limit = offset + byteCount; while (offset < limit)
{ Segment tail = writableSegment(1); int toCopy
= Math.min(limit - offset, Segment.SIZE - tail.limit);
 System.arraycopy(source, offset, tail.data,
tail.limit, toCopy); offset += toCopy;
 tail.limit += toCopy; } size += byteCount;
 return this; } Segment writableSegment(int
minimumCapacity) { if (minimumCapacity < 1 || minimumCapacity > Segment.SIZE) throw new
IllegalArgumentException(); if (head == null) {
head = SegmentPool.take(); // Acquire a first segment.
 return head.next = head.prev = head; } Segment
tail = head.prev; if (tail.limit + minimumCapacity >
Segment.SIZE || !tail.owner) { tail =
tail.push(SegmentPool.take()); // Append a new empty
segment to fill up. } return tail; }
```

核心逻辑是缓存大小的管理，然后调用System.arraycopy将数据复制到容器中。

## 4、数据的压缩加密

对这一部分的处理主要在BridgeInterceptor中。会在头部自动添加Accept-Encoding: gzip，并自动对response的进行解压缩，若手动添加了，则不处理response的数据。

对于发送数据的body，[官方推荐自定义拦截器实现](#)。拦截器内选用Gzip或者其他压缩算法对数据进行压缩。

除了单纯的压缩，使用protobuf代替json也是一种选择，除了压缩率和速度，protobuf对数据是一种天然的混淆，更安全一些，但是使用起来比json要麻烦。

同样的手段，也可以插入一个自定义的拦截器来对数据进行加密。

## 网络请求的监控

okhttp在3.11版本开始提供了一个网络时间监控的回调接口

[HttpEventListener](#)，能进行一些耗时和事件统计。

[360的方案](#)加入拦截器统计响应时间和上下行流量。

## 网络质量的监控

okhttp没有这部分内容，但是有一些[工具](#)可以用，可以执行linux的ping命令，在socket连接前后加入计时，使用traceroute（利用ICMP协议来查看到目标机器链路中的节点的可达性，其报文内会含有目标网络、主机、端口可达性等一系列信息，再加上ip协议的TTL来遍历当前节点到目标节点的链路信息），程序实现参考[《traceroute程序-c语言实现》](#)

## 弱网优化和失败处理

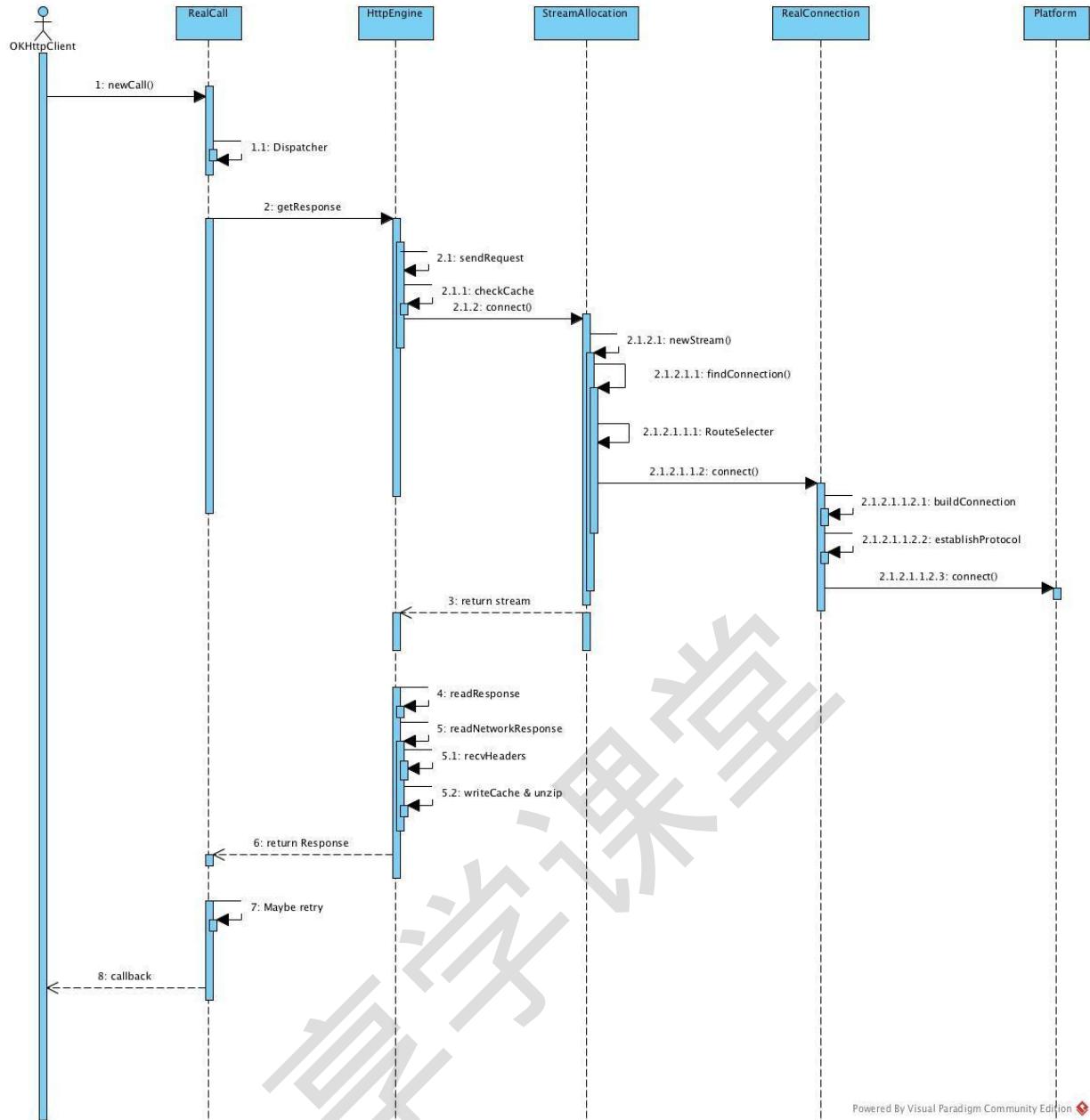
这部分就留坑吧。okhttp对网络失败做了处理，但是说到针对弱网的优化，还是要去翻看mars。

# 第十二节OkHttp面试之--OkHttp的整个异步请求流

通过上一节，我们已经了解了如何使用OkHttp发送异步请求，下载网络图片信息并显示到ImageView控件上，从这一节开始我们就来开始研究一下其内部的实现流程和原理。因为整个流程相对而言还是比较复杂，因此对于流程的分析我划分成以下几个章节去介绍

1. 流程概述
2. 拦截器的原理
3. HttpEngine中sendRequest的流程分析
4. HttpEngine中readResponse的流程分析

这一节我们先来看一下整个流程的概述，先上一张时序图



以上图片来自<http://www.jianshu.com/p/db197279f053>

就从上一些OkHttp的使用开始分析：

上一节我们通过OkHttpClient.newCall(Request)的方法创建出一个Call对象，

Call是一个接口代码如下：

```
package okhttp3; import java.io.IOException; public interface Call { Request request(); Response execute() throws IOException; void enqueue(Callback responseCallback); void cancel(); boolean isExecuted(); boolean isCanceled(); interface Factory { Call newCall(Request request); } }
```

因此OkHttpClient需要给我们返回一个Call接口的实现类，点击去看一下，代码如下：

```
/** * Prepares the {@code request} to be executed at some point in the future. */ @Override public Call newCall(Request request) { return new RealCall(this, request); }
```

可以发现，其实返回的是RealCall对象。

接下来在Activity中调用Call.enqueue(Callback)方法，点进去之后代码如下：

```
@Override public void enqueue(Callback responseCallback) { enqueue(responseCallback, false); }
void enqueue(Callback responseCallback, boolean forWebSocket) { synchronized (this) { if (executed) throw new IllegalStateException("Already Executed"); executed = true; } client.dispatcher().enqueue(new AsyncCall(responseCallback, forWebSocket)); }
```

在enqueue一参的方法中又调用了2参数的方法，并将第二个参数forWebSocket置为了false。

而在2参数的enqueue方法中，我们发现最终调用了client.dispatcher.enqueue的方法，并且将Callback和forWebSocket两个参数封装到了一个叫做AsyncCall的对象当中。

这个dispatcher就是OkHttpClient中的一个全局变量，点进去查看如下：

```
synchronized void enqueue(AsyncCall call) { if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) { runningAsyncCalls.add(call); executorService().execute(call); } else { readyAsyncCalls.add(call); } }
```

Dispatcher内部是过一个线程池来执行，超过最大请求数后则先加入准备请求的队列中，对于这个线程池后续会单独用一篇博客来讲解，此时我们主要来整理一下整个请求流程，因此不做过多解释。以上代码会执行到if代码块，然后执行executorService().execute(call); 看到这对于线程池有

了解的同学应该能立马想到AsyncCall其实就是一个Runnable对象，我们点进AsyncCall代码中可以看到它就是继承了NamedRunnable，而NamedRunnable又实现了Runnable接口并且在run方法中主动调用了execute的方法，如下所示：

```
1 /**
2 * public abstract class NamedRunnable implements Runnable {
3 * protected final String name;
4 *
5 * public NamedRunnable(String format, Object... args) { this.name = String.format(format, args); }
6 *
7 * @Override public final void run() {
8 * String oldName = Thread.currentThread().getName();
9 * Thread.currentThread().setName(name);
10 * try {
11 * execute();
12 * } finally {
13 * Thread.currentThread().setName(oldName);
14 * }
15 * }
16 *
17 * protected abstract void execute();
18 }
```

而这个抽象的execute方法在AsyncCall中已经被实现，如下所示：

```
1 @Override protected void execute() {
2 boolean signalledCallback = false;
3 try {
4 Response response = getResponseWithInterceptorChain(forWebSocket);
5 if (canceled) {
6 signalledCallback = true;
7 responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
8 } else {
9 signalledCallback = true;
10 responseCallback.onResponse(RealCall.this, response);
11 }
12 } catch (IOException e) {
13 if (signalledCallback) {
14 // Do not signal the callback twice!
15 logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);
16 } else {
17 responseCallback.onFailure(RealCall.this, e);
18 }
19 } finally {
20 client.dispatcher().finished(this);
21 }
22 }
```

由上图可以看到，当执行一个AsyncCall的execute方法时，会调用一个叫做getResponseWithInterceptorChain的方法，并返回Response对象，然后通过接口回调的方式将此Response返回给我们在Activity中所实现的Callback接口并刷新UI。

那这个getResponseWithInterceptorChain内部又是如何工作的呢？？其实这个方法中的源码很简单，如下所示：

```
1 private Response getResponseWithInterceptorChain(boolean forWebSocket) throws IOException {
2 Interceptor.Chain chain = new ApplicationInterceptorChain(0, originalRequest, forWebSocket);
3 return chain.proceed(originalRequest);
4 }
```

创建了一个ApplicationInterceptorChain对象，然后调用其proceed方法进行，接下来看一下ApplicationInterceptorChain的proceed代码如下

```
@Override public Response proceed(Request request) throws IOException {
 // If there's another interceptor in the chain, call that.
 if (index < client.interceptors().size()) {
 Interceptor.Chain chain = new ApplicationInterceptorChain(index + 1, request, forWebSocket);
 Interceptor interceptor = client.interceptors().get(index);
 Response interceptedResponse = interceptor.intercept(chain);

 if (interceptedResponse == null) {
 throw new NullPointerException("application interceptor " + interceptor
 + " returned null");
 }
 return interceptedResponse;
 }
 // No more interceptors. Do HTTP.
 return getResponse(request, forWebSocket);
}
```

循环递归的方式遍历OkHttpClient中所有的拦截器，并调用其intercept方法

在此方法中进行真正的网络请求，并返回Response对象

在proceed方法中，判断如果index小于OkHttpClient中拦截器集合的个数，则会递归创建新的ApplicationInterceptorChain对象，并将这个新的ApplicationInterceptorChain对象那个传递给index下标的拦截器的intercept方法

**注意：此处有一个亮点，如果我们想自己实现Intercept一定要在intercept方法中主动调用chain.proceed方法，这样整个递归循环才能顺利的执行下去，反过来说我们也可以在某一个拦截器中将网络请求进行拦截，做法就是只要在拦截器中不调用proceed方法即可**

**对于拦截器的作用以及原理我会在后续章节中单独分析**

但是在这个递归循环中，最终index都会处于一个 $\geq \text{client.interceptors.size}$ 的阶段，因此最终会调用最后一行代码getResponse(request, forWebSocket)方法。这个方法是真正的发送网络请求并获取Response的方法，其内部代码如下所示：

```
Response getResponse(Request request, boolean forWebSocket)
throws IOException {

 ...

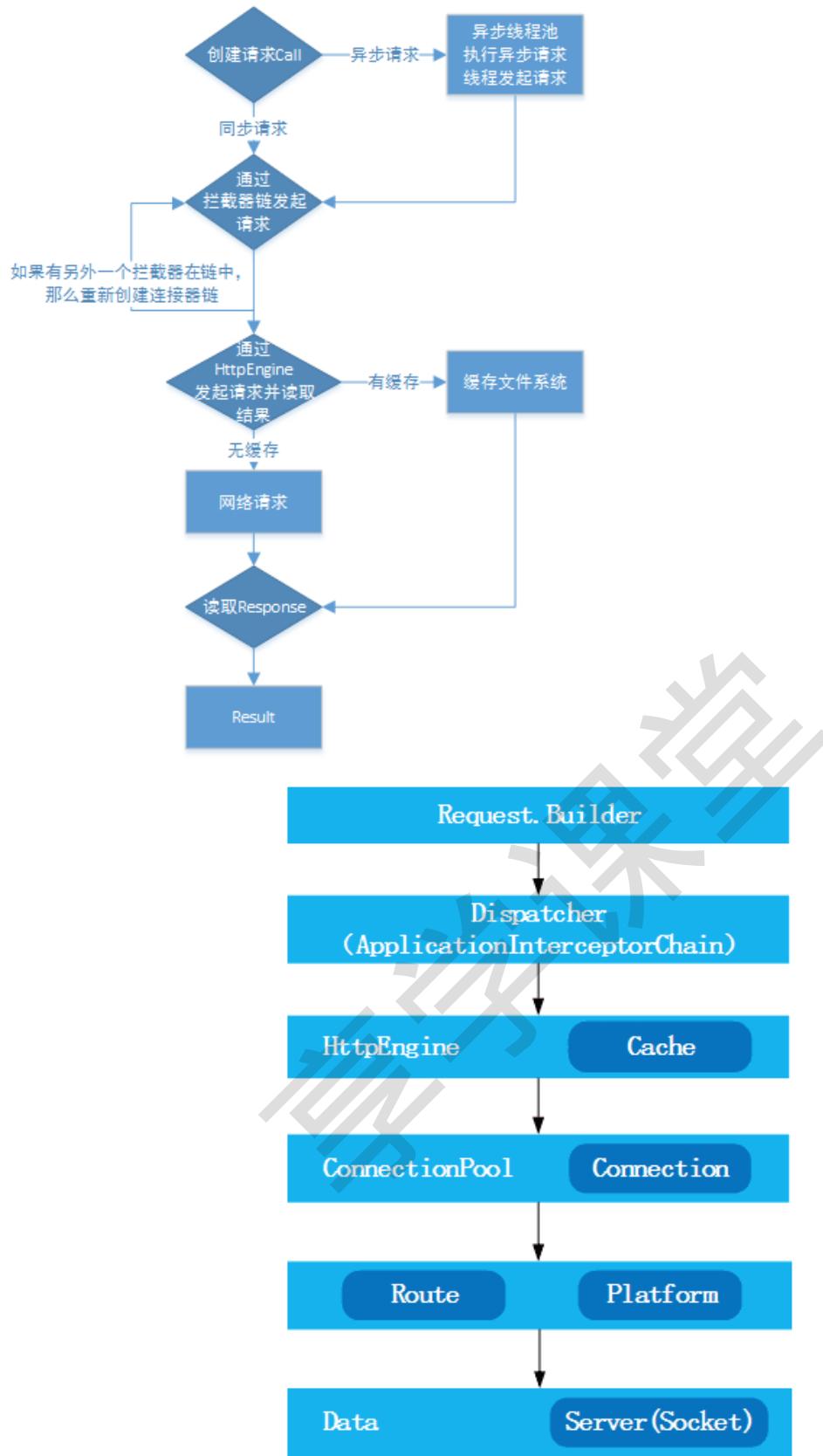
 engine = new HttpEngine(client, request, false, false,
forWebSocket, null, null, null);

 int followUpCount = 0;
 while (true) {
 if (canceled) {
 engine.releaseStreamAllocation();
 throw new IOException("Canceled");
 }
 }发送网络请求，获取请求结果对象
 boolean releaseConnection = true;
 try {
 engine.sendRequest();
 engine.readResponse();
 releaseConnection = false;
 } catch {
 ...
 }
}
```

如图所示，在getResponse方法中创建了一个HttpEngine的对象，然后分别调用HttpEngine的sendRequest和readResponse方法，这两个方法依次是发送网络请求和读取网络请求结果，最后将Response对象返回到之前说的getResponseWithInterceptorChain方法，并回传给Activity中的Callback

对于HttpEngine内部实现后续会单独再做分析。

**最后用两个图来做一个流程总结**



下一节我们一起来看一下HttpEngine.sendRequest方法内部是如何处理的

## 第十三节 OkHttp面试之--HttpEngine中的sendRequest方法详解

上一节我们介绍了OkHttp网络异步请求的整个流程。其中在流程的最后阶段，我们发现最终创建了HttpEngine对象，并分别调用的此对象的sendRequest和readResponse方法。这两个方法 分别有它相应的作用。这一节我们着重来分析sendRequest流程。

以下是sendRequest的整个方法中的内容：

面试课堂

想  
識  
思  
學

```
public void sendRequest() throws RequestException,
RouteException, IOException { if (cacheStrategy != null) return; // Already sent. if (httpStream != null)
throw new IllegalStateException(); Request request =
networkRequest(userRequest); InternalCache
responseCache = Internal.instance.internalCache(client);
 Response cacheCandidate = responseCache != null
? responseCache.get(request) : null; long now =
System.currentTimeMillis(); cacheStrategy = new
CacheStrategy.Factory(now, request,
cacheCandidate).get(); networkRequest =
cacheStrategy.networkRequest; cacheResponse =
cacheStrategy.cacheResponse; if (responseCache != null) {
 responseCache.trackResponse(cacheStrategy);
 } if (cacheCandidate != null && cacheResponse == null) {
 closeQuietly(cacheCandidate.body()); // The
cache candidate wasn't applicable. Close it. } //
If we're forbidden from using the network and the cache
is insufficient, fail. if (networkRequest == null &&
cacheResponse == null) {
 userResponse = new
Response.Builder()
 .request(userRequest)
 .priorResponse(stripBody(priorResponse))
 .protocol(Protocol.HTTP_1_1)
 .code(504)
 .message("Unsatisfiable Request (only-if-cached)")
 .body(EMPTY_BODY)
 .build();
 return;
 }
 // If we don't need the network, we're done. if
(networkRequest == null) {
 userResponse =
cacheResponse.newBuilder()
 .request(userRequest)
 .priorResponse(stripBody(priorResponse))
 .cacheResponse(stripBody(cacheResponse))
 .build();
 userResponse = unzip(userResponse);
 return;
 } // We need the network to satisfy this
request. Possibly for validating a conditional GET.
boolean success = false; try {
 httpStream =
connect();
 httpStream.setHttpEngine(this);
 if
(writeRequestHeadersEagerly()) {
 long
contentLength = OkHeaders.contentLength(request);
 if (bufferRequestBody) {
 if (contentLength >
```

```
Integer.MAX_VALUE) { throw new
IllegalStateException("Use setFixedLengthStreamingMode()
or "
+ "setChunkedStreamingMode() for
requests larger than 2 GiB."); } if
(contentLength != -1) { // Buffer a request
body of a known length.
httpStream.writeRequestHeaders(networkRequest);
requestBodyOut = new RetryableSink((int) contentLength);
} else { // Buffer a request body of
an unknown length. Don't write request headers until the
// entire body is ready; otherwise we can't
set the Content-Length header correctly.
requestBodyOut = new RetryableSink(); }
} else {
httpStream.writeRequestHeaders(networkRequest);
requestBodyOut =
httpStream.createRequestBody(networkRequest,
contentLength); } success = true; }
finally { // If we're crashing on I/O or otherwise,
don't leak the cache body. if (!success &&
cacheCandidate != null) {
closeQuietly(cacheCandidate.body()); } }
```

可以看到sendRequest方法比较长，我对它进行分块解析。其中主要分以下两大块

1 先从 Cache 中判断当前请求是否可以从缓存中返回

```

if (cacheStrategy != null) return; // Already sent.
if (httpStream != null) throw new IllegalStateException();

Request request = networkRequest(userRequest);

InternalCache responseCache = Internal.instance.internalCache(client);
Response cacheCandidate = responseCache != null
 ? responseCache.get(request)
 : null;
long now = System.currentTimeMillis();
cacheStrategy = new CacheStrategy.Factory(now, request, cacheCandidate).get();
networkRequest = cacheStrategy.networkRequest;
cacheResponse = cacheStrategy.cacheResponse;

if (responseCache != null) {
 responseCache.trackResponse(cacheStrategy);
}

if (cacheCandidate != null && cacheResponse == null) {
 closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close it.
}

```

□

读取所有的请求缓存

从缓存中读取之前相同请求得到的Response

根据请求和缓存结果（可能为null）去得到缓存策略  
如果请求策略为只要缓存则  
networkRequest,cacheResponse都为空，大部分条件都会得到networkRequest就是request，  
cacheResponse为空

## 2 如果没有Cache则连接网络

```

// We need the network to satisfy this request. Possibly for validating a conditional GET.
boolean success = false;
try {
 httpStream = connect(); 通过connect()方法获取HttpStream对象
 httpStream.setHttpEngine(this); HttpStream中封装了Socket请求,根据http1.0协议传输数据
}

```

先来看下connect方法中是如何创建HttpStream对象的

```

private HttpStream connect() throws RouteException, RequestException, IOException {
 boolean doExtensiveHealthChecks = !networkRequest.method().equals("GET");
 return streamAllocation.newStream(client.connectTimeoutMillis(),
 client.readTimeoutMillis(), client.writeTimeoutMillis(),
 client.retryOnConnectionFailure(), doExtensiveHealthChecks);
}

```

调用SteamAllocation.newStream的方法创建HttpStream对象并返回。  
点进去代码如下所示：

```
public HttpStream newStream(int connectTimeout, int readTimeout, int writeTimeout,
 boolean connectionRetryEnabled, boolean doExtensiveHealthChecks)
 throws RouteException, IOException {
try {
 RealConnection resultConnection = findHealthyConnection(connectTimeout, readTimeout,
 writeTimeout, connectionRetryEnabled, doExtensiveHealthChecks);
 // 查找可用的Socket对象

 HttpStream resultStream; 根据Connection对象获取相应的HttpStream对象
 if (resultConnection.framedConnection != null) {
 resultStream = new Http2xStream(this, resultConnection.framedConnection);
 } else {
 resultConnection.socket().setSoTimeout(readTimeout);
 resultConnection.source.timeout().timeout(readTimeout, MILLISCONDS);
 resultConnection.sink.timeout().timeout(writeTimeout, MILLISCONDS);
 resultStream = new Http1xStream(this, resultConnection.source, resultConnection.sink);
 }
 // 根据Connection对象获取相应的HttpStream对象

 synchronized (connectionPool) {
 stream = resultStream;
 return resultStream;
 }
} catch (IOException e) {
 throw new RouteException(e);
}
}
```

从上图中可以看出，在newStream方法中先通过findHealthyConnection方法获取一个RealConnection对象，实际上就是查找可用的Socket对象。在OkHttp框架中有一个特点就是OkHttp可以使用一个Socket对象来维护拥有过个ip的Server端，对于Socket的实现后续再单独讲解，此处不再做介绍。

获取RealConnction对象之后，根据此对象再获取相应的HttpStream对象，我们一般返回的是Http1xStream对象，最后将resultStream赋值给全局变量stream。而这个全局变量会在下一节readResponse方法中再使用。

注意：本节主要对于sendRequest方法中比较核心的代码进行的跟踪分析，在此方法中还有对Request请求的Head和Body的添加操作并没有进行详细描述。感兴趣的同学可以自行研究。

下一节继续讲解HttpEngine.readResponse方法的流程

## 第十四节 OkHttp解析大总结

先来看一下Request这个类，此类还是比较简单的，没有太多复杂的代码，内部就是对一些url、method、body等变量的封装。在整个OkHttp的网络请求流程中有一大部分的代码就是用来对这几个变量进行二次修改。源码如下：

```

} */

public final class Request {
 private final HttpUrl url;
 private final String method;
 private final Headers headers;
 private final RequestBody body;
 private final Object tag;

 private volatile URI javaNetUri; // Lazily initialized.
 private volatile CacheControl cacheControl; // Lazily initialized.

 private Request(Builder builder) {
 this.url = builder.url;
 this.method = builder.method;
 this.headers = builder.headers.build();
 this.body = builder.body;
 this.tag = builder.tag != null ? builder.tag : this;
 }
}

```

接着再来看Call这个类，在[OkHttp面试之-OkHttp的整个异步请求流程](#)这一节中我们了解到真正的Call对象应该是RealCall类型，那我们就来看一下这个RealCall

```

final class RealCall implements Call {
 private final OkHttpClient client;

 // Guarded by this.
 private boolean executed;
 volatile boolean canceled;

 /** The application's original request unadulterated by redirects or auth headers. */
 Request originalRequest;
 HttpEngine engine;

 protected RealCall(OkHttpClient client, Request originalRequest) {
 this.client = client;
 this.originalRequest = originalRequest;
 }
}

```

从上面的构造器中我们能看出，其内部也保存以一个OkHttpClient和一个Request对象的实例。并且内部还有一个HttpEngine的全局变量，对于HttpEngine目前阶段我们可以将其理解为网络访问的入口—通过HttpEngine进行网络访问操作并返回请求结果。

那在RealCall中是如何使用HttpEngine进行网络访问的呢？继续分析代码看到在RealCall.enqueue(Callback)方法执行后续流程

```

void enqueue(Callback responseCallback, boolean forWebSocket) {
 synchronized (this) {
 if (executed) throw new IllegalStateException("Already Executed");
 executed = true;
 }
 client.dispatcher().enqueue(new AsyncCall(responseCallback, forWebSocket));
}

```

继续

```

 }
 client.dispatcher().enqueue(new AsyncCall(responseCallback, forWebSocket));
}

@Override public void cancel() {
 canceled = true;
 if (engine != null) engine.cancel();
}

@Override public synchronized boolean isExecuted() { return executed; }

@Override public boolean isCanceled() { return canceled; }

final class AsyncCall extends NamedRunnable {
 private final Callback responseCallback;
 private final boolean forWebSocket;

 private AsyncCall(Callback responseCallback, boolean forWebSocket) {
 super("OkHttp %s", originalRequest.url().toString());
 this.responseCallback = responseCallback; → 网络请求的接口回调, 由开发者自己实现
 this.forWebSocket = forWebSocket; → false
 }
}

```

AsyncCall是RealCall中的一个内部类，它就是一个线程我们跟踪其run方法发现其内部最终调用了自身的execute方法，代码如下：

```

@Override protected void execute() {
 boolean signalledCallback = false; → 在获得网络请求的Response之前, 先经过拦截链的洗礼
 try {
 Response response = getResponseWithInterceptorChain(forWebSocket);
 if (canceled) {
 signalledCallback = true;
 responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
 } else {
 signalledCallback = true;
 responseCallback.onResponse(RealCall.this, response); → 将请求结果通过接口回调返回给Activity
 }
 } catch (IOException e) {
 if (signalledCallback) {
 // Do not signal the callback twice!
 logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);
 } else {
 responseCallback.onFailure(RealCall.this, e);
 }
 } finally {
 client.dispatcher().finished(this);
 }
}

```

用脚后跟想了一下关键的代码肯定是在  
getResponseWithInterceptorChain这个方法中了吧,继续跟下去吧。。。  
顺便看下这个拦截器“揪净”有几根头发!

```

private Response getResponseWithInterceptorChain(boolean forWebSocket) throws IOException {
 Interceptor.Chain chain = new ApplicationInterceptorChain(0, originalRequest, forWebSocket);
 return chain.proceed(originalRequest);
}

class ApplicationInterceptorChain implements Interceptor.Chain {
 private final int index;
 private final Request request;
 private final boolean forWebSocket;

 ApplicationInterceptorChain(int index, Request request, boolean forWebSocket) {
 this.index = index;
 this.request = request;
 this.forWebSocket = forWebSocket;
 }

 @Override public Connection connection() { return null; }

 @Override public Request request() {
 return request;
 }

 @Override public Response proceed(Request request) throws IOException {
 // If there's another interceptor in the chain, call that.
 if (index < client.interceptors().size()) {
 Interceptor.Chain chain = new ApplicationInterceptorChain(index + 1, request, forWebSocket);
 Interceptor interceptor = client.interceptors().get(index);
 Response interceptedResponse = interceptor.intercept(chain);

 if (interceptedResponse == null) {
 throw new NullPointerException("application interceptor " + interceptor

```

在`getResponseWithInterceptorChain`中新建了一个`ApplicationInterceptorChain`对象，并调用其`proceed`方法，并且传入的`originalRequest`就是我们在Activity中所初始化的`Request`对象。

接下来我们来看看这个`ApplicationInterceptorChain`是什么。从上图中我们看到它继承了`Interceptor.Chain`类，点击去看一下：

```

public interface Interceptor { Response intercept(Chain
chain) throws IOException; interface Chain { Request
request(); Response proceed(Request request) throws
IOException; Connection connection(); }}

```

可以看到`Interceptor`是一个接口，内部只有一个`intercept`方法，用来处理该拦截器想要如何处理请求。而`Chain`是`Interceptor`中的一个内部接口，它里面最重要的方法就是`proceed`方法。接下来我们回到`ApplicationInterceptorChain`中看一下它是如何实现`proceed`方法的

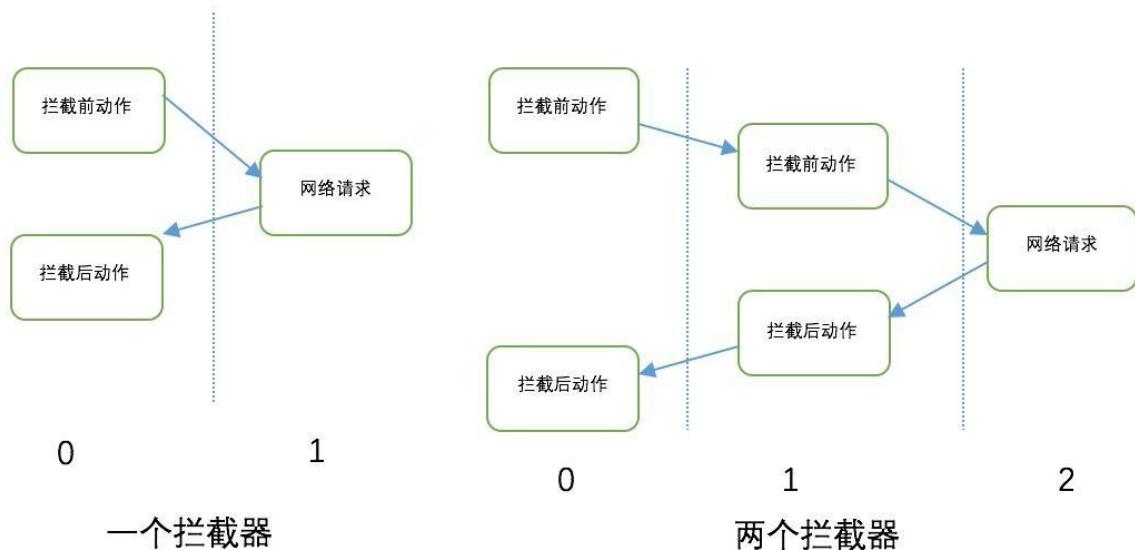
```
@Override public Response proceed(Request request) throws
IOException { // If there's another interceptor in
the chain, call that. if (index <
client.interceptors().size()) { Interceptor.Chain
chain = new ApplicationInterceptorChain(index + 1,
request, forWebSocket); Interceptor interceptor =
client.interceptors().get(index); Response
interceptedResponse = interceptor.intercept(chain);
if (interceptedResponse == null) { throw new
NullPointerException("application interceptor " +
interceptor + " returned null"); }
return interceptedResponse; } // No more
interceptors. Do HTTP. return getResponse(request,
forWebSocket); }
```

index默认为0. 而client.interceptors()返回的是OkHttpClient内部的一个全局List变量，如下所示

```
final Dispatcher dispatcher;
final Proxy proxy;
final List<Protocol> protocols;
final List<ConnectionSpec> connectionSpecs;
final List<Interceptor> interceptors; // 目标
final List<Interceptor> networkInterceptors;
final ProxySelector proxySelector;
final CookieJar cookieJar;
final Cache cache;
```

我们可以通过OkHttpClient.addInterceptor(Interceptor对象)的方式向这个集合中添加拦截器对象。一旦集合中有拦截器对象，则会在此方法中进行循环递归的方式遍历client中所有的intercepts(请求前拦截). 在递归遍历的最里层最终调用了最后一行代码getResponse方法获取请求结果，再层层往回传递。

用一张图形象的描述一下过程及就是如下图所示：



**注意：当我们在自己实现`Interceptor`时并复写`intercept`方法时，一定要记住调用`chain.proceed`方法，否则上面提到的循环递归则会终止，也就是说最终真正的发送网络请求并获取结果的`getResponse`方法就不会被调用**

既然说到拦截器了，我们就来看看OkHttp中的拦截器究竟是个什么玩意。

拦截器是一种强大的机制，可以监视、重写和重试调用。我们可以用拦截器做很多事情，添加我们自己的头部信息，设置有网请求，没网走缓存等。

拦截器分为两种拦截器，一种是应用拦截器，一种是网络拦截器。

这两种拦截器的区别主要是以下几点：

网络拦截器和应用拦截器的区别主要有以下几点：

应用拦截器： 1. 不需要担心中间过程的响应，如重定向和重试。

2. 总是只调用一次，即使HTTP响应是从缓存中获取。 3. 观察应用程序的初衷。不关心OkHttp注入的头信息如：`If-None-Match`。 4. 允许短路而不调用 `chain.proceed()`，即中止调用。 5. 允许重试，使 `Chain.proceed()` 调用多次。
- 网络拦截器： 1. 能够操作中间过程的响应，如重定向和重试。 2. 当网络短路而返回缓存响应时不被调用。 3. 只观察在网络上传输的数据。 4. 携带请求来访问连接。

至于拦截器的具体使用场景，可以参考[Picasso高逼格使用技巧](#)这篇文章中的如下代码

```
private static OkHttpClient getProgressBarClient(final ProgressListener listener) { return getHttpClient().newBuilder().addNetworkInterceptor(new Interceptor() { @Override public Response intercept(Chain chain) throws IOException { Response originalResponse = chain.proceed(chain.request()); return originalResponse.newBuilder() .body(new ProgressResponseBody(originalResponse.body(), listener)) .build(); } }).build(); }
```

好了回到正题上，刚才说了在循环递归遍历的最里层，最终会调用getResponse方法发送网络请求并返回Response对象

想  
識  
思  
學

```
Response getResponse(Request request, boolean
forWebSocket) throws IOException { // Copy body
metadata to the appropriate request headers.
RequestBody body = request.body(); if (body != null) {
 Request.Builder requestBuilder =
request.newBuilder(); MediaType contentType =
body.contentType(); if (contentType != null) {
requestBuilder.header("Content-Type",
contentType.toString()); } long contentLength =
body.contentLength(); if (contentLength != -1) {
 requestBuilder.header("Content-Length",
Long.toString(contentLength));
requestBuilder.removeHeader("Transfer-Encoding");
} else { requestBuilder.header("Transfer-Encoding",
"chunked"); requestBuilder.removeHeader("Content-
Length"); } request = requestBuilder.build();
} // Create the initial HTTP engine. Retries and
redirects need new engine for each attempt. engine =
new HttpEngine(client, request, false, false,
forWebSocket, null, null, null); int followUpCount =
0; while (true) { if (canceled) {
engine.releaseStreamAllocation(); throw new
IOException("Canceled"); } boolean
releaseConnection = true; try {
engine.sendRequest(); engine.readResponse();
releaseConnection = false; } catch (RequestException
e) { // The attempt to interpret the request
failed. Give up. throw e.getCause(); } catch
(RouteException e) { // The attempt to connect via
a route failed. The request will not have been sent.
HttpEngine retryEngine =
engine.recover(e.getLastConnectException(), null);
if (retryEngine != null) { releaseConnection =
false; engine = retryEngine; continue;
} // Give up; recovery is not possible.
throw e.getLastConnectException(); } catch
(IOException e) { // An attempt to communicate
with a server failed. The request may have been sent.
```

```
HttpEngine retryEngine = engine.recover(e, null);
if (retryEngine != null) { releaseConnection
= false; engine = retryEngine;
continue; } // Give up; recovery is not
possible. throw e; } finally { //
We're throwing an unchecked exception. Release any
resources. if (releaseConnection) {
StreamAllocation streamAllocation = engine.close();
streamAllocation.release(); }
Response response = engine.getResponse(); Request
followUp = engine.followUpRequest(); if (followUp ==
null) { if (!forWebSocket) {
engine.releaseStreamAllocation(); } return
response; } StreamAllocation streamAllocation =
engine.close(); if (++followUpCount >
MAX_FOLLOW_UPS) { streamAllocation.release();
throw new ProtocolException("Too many follow-up
requests: " + followUpCount); } if
(!engine.sameConnection(followUp.url())) {
streamAllocation.release(); streamAllocation =
null; } request = followUp; engine = new
HttpEngine(client, request, false, false, forWebSocket,
streamAllocation, null, response); } }
```

## 以上代码比较长，我们分段来解释

- 首先看第3行到21行是根据request中的参数重新构建一个request对象，在Request的头部加入了一些参数
- 然后在25行根据request和OkHttpClient创建了一个HttpEngine，然后调用此HttpEngine.sendRequest方法发送请求以及readResponse方法读取获取的请求结果信息。
- 在第44行和54行如果访问时发生了错误，例如RouteException、IOException，则会尝试重连，调用HttpEngine的recover方法，重新生成一个HttpEngine来访问，然后重新进入刚刚的循环
- 在第78行如果访问成功，并且不需要重定向，则将在第71行通过getResponse方法来获取的请求结果返回

- 在第72行调用followUpRequest来获取重定向请求，如果此重定向请求不为null，则代码会执行从81行~96行，主要是重新创建一个HttpEngine对象再进行请求操作。这里需要了解下重定向的概念
  - 一个简单的重定向例子就是一个网站，登陆的用户分为两种：管理员和游客，同一个url，对于不同身份的人来说访问到的页面可能不一样，这里就是用了重定向功能，例如该网站的某个页面，如果管理员访问是长某某样子（网页A），游客访问是长某某样子（网页B），这两个网页对应的url也不同，但是初始访问的url都是一样的，这是通过服务器对url和用户解析，返回一个新的url让用户去访问。不知道这么讲大家懂了没。。。一个简单的请求，重定向可以多次，不同的浏览器支持不同的次数。OkHttp框架中重定向最大次数由HttpEngine.MAX\_FOLLOW\_UPS决定：

```
public final class HttpEngine {
 /**
 * How many redirects and auth challenges should we attempt? Chrome follows 21 redirects; Firefox,
 * curl, and wget follow 20; Safari follows 16; and HTTP/1.0 recommends 5.
 */
 public static final int MAX_FOLLOW_UPS = 20;
```

上面这段代码基本讲解完毕，但是在代码中，还看到一个概念StreamAllocation，对于StreamAllocation我将专门重新写一篇文章进行讲解-[OkHttp中StreamAllocation的分析](#) 这篇文章一定要看！！在了解了什么是StreamAllocation、Http和Socket相关知识之后，我们重新回到刚才的代码中的下列片段，也就是88~95行

```
if (!engine.sameConnection(followUp.url())) {
 streamAllocation.release(); streamAllocation =
 null;} request = followUp; engine = new
 HttpEngine(client, request, false, false, forWebSocket,
 streamAllocation, null, response);
```

当发现需要重定向的时候，就会执行这段代码，首先先检测重定向的url和刚刚的请求是不是同一个Connection，看下sameConnection函数：

```
public boolean sameConnection(HttpUrl followUp) {
 HttpUrl url = userRequest.url(); return
 url.host().equals(followUp.host()) && url.port()
 == followUp.port() &&
 url.scheme().equals(followUp.scheme());}
```

这函数很简单，只是看下这两个url的host、port、scheme是不是一样。如果发现不一样，就释放HttpEngine原来的streamAllocation，并置空，如果发现一样，则重用刚刚的stream。HttpEngine的构造函数里面会判断传入的StreamAllocation是不是为空，若为空则创建一个根据request，并传入ConnectionPool，创建一个streamAllocation，并且从ConnectionPool中取出Connection，并将该Connection记录到StreamAllocation中，如果没有可用的RealConnection，就创建个新的，然后再放到ConnectionPool中

## 重点开始放到HttpEngine身上

先从followUpRequest()方法下手：

client发送一个request之后，server可能回复一个重定向的response，并在这个response中告知client需要重新访问的server的IP。此时，client需要重新向新的server发送request，并等待新server的回复。所以我们需要单独判断重定向response，并发送多次request。有了OKHttp，这一切你都不用管，它会自动帮你完成所有这一切。OKHttp中followUpRequest()方法就是完成这个功能的。是不是瞬间感觉OKHttp强大到不要不要的啊~。这个方法流程比较简单，就不给出流程图了

想  
識  
思  
學

```
public Request followUpRequest() throws IOException {
 if (userResponse == null) throw new
IllegalStateException(); Proxy selectedProxy =
getRoute() != null ? getRoute().getProxy()
: client.getProxy(); int responseCode =
userResponse.code(); // 利用responseCode来分析是否需要自动
发送后续request switch (responseCode) { // 未认证用
户, 不能访问server或代理, 故需要发送认证的request case
HTTP_PROXY_AUTH: if (selectedProxy.type() !=
Proxy.Type.HTTP) { throw new
ProtocolException("Received HTTP_PROXY_AUTH (407) code
while not using proxy"); } case
HTTP_UNAUTHORIZED: return
OkHeaders.processAuthHeader(client.getAuthenticator(),
userResponse, selectedProxy); // 永久重定向, 暂时重定
向, 永久移动了等和重定向相关的response, response code中以3打头的
都是 // 它们需要重新发送request给新的server, 新server的ip在
response中会给出 case HTTP_PERM_REDIRECT: case
HTTP_TEMP_REDIRECT: if
(!userRequest.method().equals("GET") &&
!userRequest.method().equals("HEAD")) { return
null; } case HTTP_MULT_CHOICE: case
HTTP_MOVED_PERM: case HTTP_MOVED_TEMP: case
HTTP_SEE_OTHER: // Does the client allow
redirects? if (!client.getFollowRedirects())
return null; // 新的server的IP地址在response的
Location header中给出 String location =
userResponse.header("Location"); if (location ==
null) return null; HttpUrl url =
userRequest.httpUrl().resolve(location); // Don't
follow redirects to unsupported protocols. if (url
== null) return null; // If configured, don't
follow redirects between SSL and non-SSL. boolean
sameScheme = url.scheme().equals(userRequest.httpUrl().scheme());
 if (!sameScheme && !client.getFollowSslRedirects())
return null; // Redirects don't include a request
body. Request.Builder requestBuilder =
```

```
userRequest.newBuilder(); if
(HttpMethod.permitsRequestBody(userRequest.method())) {
 requestBuilder.method("GET", null);
requestBuilder.removeHeader("Transfer-Encoding");
requestBuilder.removeHeader("Content-Length");
requestBuilder.removeHeader("Content-Type"); }
// 删掉用户认证信息 if (!sameConnection(url)) {
 requestBuilder.removeHeader("Authorization");
}
return requestBuilder.url(url).build();
default: return null; } }
```

接下来再看一下sendRequest，这个方法在之前的篇章中也做过简单的介绍，在这我们在重新巩固一下：



想  
識  
思  
學

```
public void sendRequest() throws RequestException,
RouteException, IOException { if (cacheStrategy != null) return; // Already sent. if (httpStream != null)
throw new IllegalStateException(); Request request =
networkRequest(userRequest); InternalCache
responseCache = Internal.instance.internalCache(client);
 Response cacheCandidate = responseCache != null
? responseCache.get(request) : null; long now =
System.currentTimeMillis(); cacheStrategy = new
CacheStrategy.Factory(now, request,
cacheCandidate).get(); networkRequest =
cacheStrategy.networkRequest; cacheResponse =
cacheStrategy.cacheResponse; if (responseCache != null) {
 responseCache.trackResponse(cacheStrategy);
 } if (cacheCandidate != null && cacheResponse == null) {
 closeQuietly(cacheCandidate.body()); // The
cache candidate wasn't applicable. Close it. } //
If we're forbidden from using the network and the cache
is insufficient, fail. if (networkRequest == null &&
cacheResponse == null) {
 userResponse = new
Response.Builder()
 .request(userRequest)
 .priorResponse(stripBody(priorResponse))
 .protocol(Protocol.HTTP_1_1)
 .code(504)
 .message("Unsatisfiable Request (only-if-cached)")
 .body(EMPTY_BODY)
 .build();
 return;
 }
 // If we don't need the network, we're done. if
(networkRequest == null) {
 userResponse =
cacheResponse.newBuilder()
 .request(userRequest)
 .priorResponse(stripBody(priorResponse))
 .cacheResponse(stripBody(cacheResponse))
 .build();
 userResponse = unzip(userResponse);
 return;
 } // We need the network to satisfy this
request. Possibly for validating a conditional GET.
boolean success = false; try {
 httpStream =
connect();
 httpStream.setHttpEngine(this);
 if
(writeRequestHeadersEagerly()) {
 long
contentLength = OkHeaders.contentLength(request);
 if (bufferRequestBody) {
 if (contentLength >
```

```
Integer.MAX_VALUE) { throw new
IllegalStateException("Use setFixedLengthStreamingMode()
or " + "setChunkedStreamingMode() for
requests larger than 2 GiB."); } if
(contentLength != -1) { // Buffer a request
body of a known length.
httpStream.writeRequestHeaders(networkRequest);
requestBodyOut = new RetryableSink((int) contentLength);
} else { // Buffer a request body of
an unknown length. Don't write request headers until the
// entire body is ready; otherwise we can't
set the Content-Length header correctly.
requestBodyOut = new RetryableSink(); } }
else {
httpStream.writeRequestHeaders(networkRequest);
requestBodyOut =
httpStream.createRequestBody(networkRequest,
contentLength); } success = true; }
finally { // If we're crashing on I/O or otherwise,
don't leak the cache body. if (!success &&
cacheCandidate != null) {
closeQuietly(cacheCandidate.body()); } } }
```

代码还是很长的，先来看下第五行，先是调用了一个networkRequest返回了一个request，这个函数就是对原来我们外部传进去的request做了一个封装，封装成一个真正访问网络请求的request。在HttpEngine中有两个request，一个叫userRequest，一个是networkRequest，第一个是外部传入的，未经OkHttp修改的，第二个是根据userRequest封装的一个request，用来访问网络。接下来就是获取InternalCache，Internal.instance是一个单例，该单例初始化是在OkHttpClient里面，可以看到调用Internal.instance.internalCache(client)函数只是调用了client(OkHttpClient)的internalCache，这个函数是返回了我们传入OkHttpClient的cache，如果你创建OkHttpClient的时候，没有传入，那么这里就会返回空。

得到了InternalCache后，尝试根据request去获取response，当然可能为空。接下来就到CacheStrategy了，CacheStrategy是一个决定访问网络还是访问缓存的类。CacheStrategy.Factory是工厂类，通过传入

request和刚刚internalCache中获取到的response，去get一个CacheStrategy，Factory.get函数主要是对request和response做了一个检查，会根据response合不合法、是否过期、request的参数来判断，最后返回一个CacheStrategy。CacheStrategy很简单，里面有两个变量，一个是networkRequest，一个是cacheResponse（这个变量和传入Factory的response可能是不一样的哦！）。返回到刚刚的HttpEngine.sendRequest函数中看第17行，如果InternalCache不为空，就调用trackResponse，这个函数很简单，就是记录下缓存读取命中率这些数据。然后如果从InternalCache中response不为空，但是cacheStrategy的response为空，则说明刚刚InternalCache取出来的response无效，则需要关掉。如果CacheStrategy的networkRequest不为空，则说明需要进行网络访问，如果cacheResponse不为空，则说明访问缓存足以。根据这个理论，下面一直到第47行的代码不成问题，就不解释了。接下来看网络访问的部分。

网络访问主要是通过HttpStream来实现，HttpStream这是一个接口，里面定义了一些函数，这些函数是负责读取网络数据、输出数据等。实现该接口的有两个类，一个是Http2xStream，一个是Http1xStream。第一个是专门负责Http2.0版本的，第二个是负责Http1.x版本的，两者内部实现机制不一样。Http2xStream是通过一个FramedConnection，至于对FramedConnection的理解，可以看前面关于Http2.0讲解的文章，看完那个你应该比较能够大概清楚的理解他了，我就不解释了，这里我也没深入去看。

而Http1xStream则是通过sink和source来实现的，至于sink和source是什么将在

[OkHttp中sink和source的分析](#) 这一篇中做详细介绍。

今天先写到这里吧，未完待续。。。接下里会跟中Http1XStream中的connect方法实现

## 第十五节 Okhttp任务队列工作原理

## 15.1概述

### 15.1.1 引言

android完成非阻塞式的异步请求的时候都是通过启动子线程的方式来解决，子线程执行完任务的之后通过handler的方式来和主线程来完成通信。无限制的创建线程，会给系统带来大量的开销。**如果在高并发的任务下，启用个线程池，可以不断的复用里面不再使用和有效的管理线程的调度和数量的管理。就可以节省系统的成本，有效的提高执行效率。**

### 15.1.2 线程池ThreadPoolExecutor

ThreadPoolExecutor是java线程创建工具。存在于  
java.util.concurrent包中。

看构造方法：

```
public ThreadPoolExecutor(int corePoolSize,
 int maximumPoolSize,
 long keepAliveTime,
 TimeUnit unit,
 BlockingQueue<Runnable> workQueue) {
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
 workQueue, Executors.defaultThreadFactory(),
 defaultHandler); }
```

- corePoolSize : 最小并发线程数。
- maximumPoolSize : 线程池中最大的线程池并发数。
- keepAliveTime : 当线程的数目大于corePoolSize时，线程的最大存活时间。
- unit : 时间单位
- BlockingQueue 工作队列

okhttp的线程池对象存在于Dispatcher类中。实例过程如下

```
public synchronized ExecutorService executorService() {
 if (executorService == null) { executorService =
 new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60,
 TimeUnit.SECONDS, new SynchronousQueue<Runnable>
(), Util.threadFactory("okhttp Dispatcher", false)); }
 return executorService; }
```

### 15.1.3 Call对象

了解源码或使用过okhttp的都知道。okhttp的操作元是Call对象。异步的实现是RealCall.AsyncCall。而AsyncCall是实现的一个Runnable接口。

```
final class AsyncCall extends NamedRunnable {}
```

所以Call本质就是一个Runnable线程操作元肯定是放进excutorService中直接启动的。

那么okhttp是如何完成线程复用和线程管理的呢。

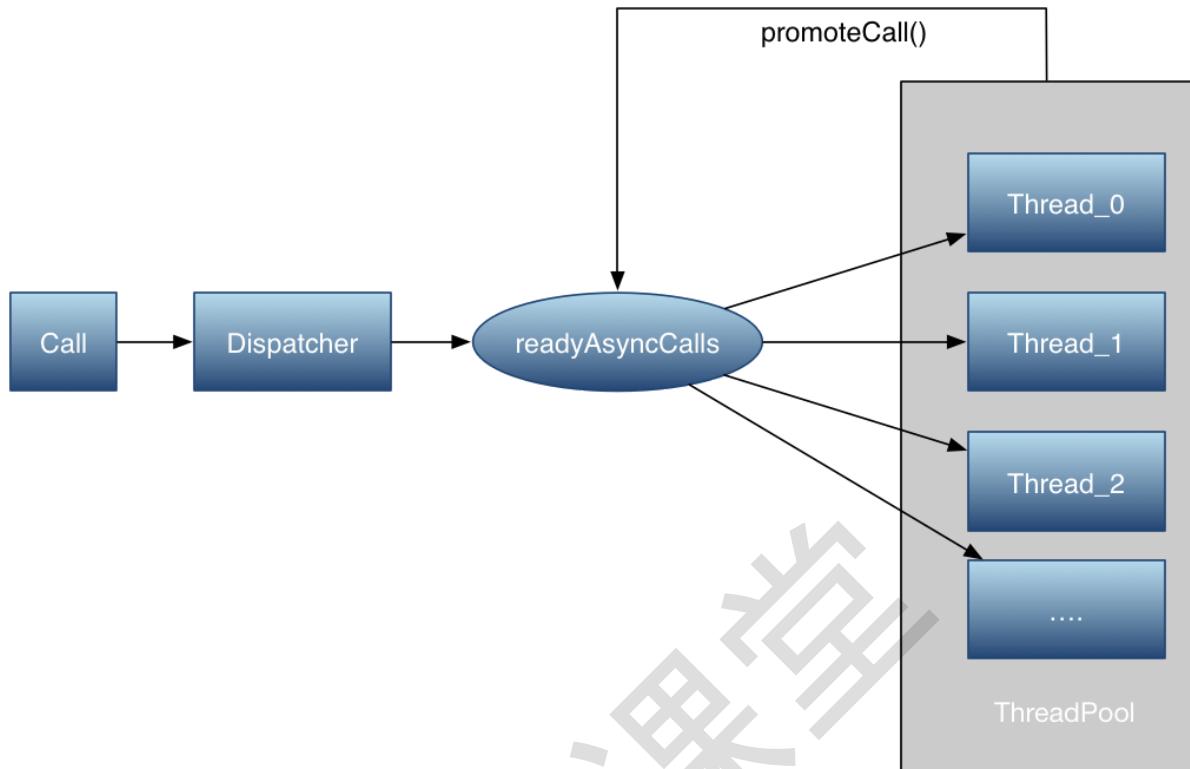
## 15.2 线程池的复用和管理

### 15.2.1 图解

为了完成调度和复用，定义了两个队列分别用作等待队列和执行任务的队列。这两个队列都是Dispatcher成员变量。Dispatcher是一个控制执行，控制所有Call的分发和任务的调度、通信、清理等操作。这里只介绍异步调度任务。

```
/** Ready async calls in the order they'll be run. */
private final Deque<AsyncCall> readyAsyncCalls = new
ArrayDeque<>(); /** Running asynchronous calls. Includes
canceled calls that haven't finished yet. */ private
final Deque<AsyncCall> runningAsyncCalls = new
ArrayDeque<>();
```

在《[okhttp连接池复用机制](#)》文章中我们在缓存Connection连接的时候也是使用的Deque双端队列。这里同样的方式，可以方便在队列头添加元素，移除尾部的元素。



### 15.2.2 过程分析

call代用enqueue方法的时候

```
synchronized void enqueue(AsyncCall call) { if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) { runningAsyncCalls.add(call); executorService().execute(call); } else { readyAsyncCalls.add(call); } }
```

方法中满足执行队列里面不足最大线程数maxRequests并且Call对应的host数目不超过maxRequestsPerHost的时候直接把call对象直接推入到执行队列里，并启动线程任务（call本质是一个Runnable）。否则，当前线程数过多，就把他推入到等待队列中。call执行完肯定需要在runningAsyncCalls队列中移除这个线程。那么readyAsyncCalls队列中的线程在什么时候才会被执行呢。

## 追溯下`AsyncCall` 线程的执行方法

```
@Override protected void execute() { boolean
signalledCallback = false; try { Response
response = getResponseWithInterceptorChain(forWebSocket);
if (canceled) { signalledCallback = true;
responseCallback.onFailure(RealCall.this, new
IOException("Canceled")); } else {
signalledCallback = true;
responseCallback.onResponse(RealCall.this, response);
} } catch (IOException e) { if
(signalledCallback) { // Do not signal the
callback twice! Platform.get().log(INFO,
"Callback failure for " + toLoggableString(), e);
} else {
responseCallback.onFailure(RealCall.this, e); }
} finally { client.dispatcher().finished(this);
} } }
```

这里做了核心request的动作，并把失败和回复数据的结果通过`responseCallback` 回调到Dispatcher。执行操作完毕了之后不管有无异常都会进入到`dispatcher`的`finished`方法。

```
private <T> void finished(Deque<T> calls, T call, boolean
promoteCalls) { int runningCallsCount; Runnable
idleCallback; synchronized (this) { if
(!calls.remove(call)) throw new AssertionError("Call
wasn't in-flight!"); if (promoteCalls)
promoteCalls(); runningCallsCount =
runningCallsCount(); idleCallback =
this.idleCallback; } if (runningCallsCount == 0 &&
idleCallback != null) { idleCallback.run(); } }
```

在这里call在runningAsyncCalls队列中被移除了，重新计算了目前正在执行的线程数量。并且调用了`promoteCalls()` 看来是来调整任务队列的，跟进去看下

```
private void promoteCalls() { if
(runningAsyncCalls.size() >= maxRequests) return; //
Already running max capacity. if
(readyAsyncCalls.isEmpty()) return; // No ready calls to
promote. for (Iterator<AsyncCall> i =
readyAsyncCalls.iterator(); i.hasNext();) {
AsyncCall call = i.next(); if
(runningCallsForHost(call) < maxRequestsPerHost) {
i.remove(); runningAsyncCalls.add(call);
executorService().execute(call); } if
(runningAsyncCalls.size() >= maxRequests) return; //
Reached max capacity. } }
```

原来实在这里对 `readyAsyncCalls` 进行调度的。最终会在 `readyAsyncCalls` 中通过 `remove` 操作把元素迭代取出并移除之后加入到 `runningAsyncCalls` 的执行队列中执行操作。`ArrayDeque` 是非线程安全的所以 `finished` 在调用 `promoteCalls` 的时候都在 `synchronized` 块中执行的。执行等待队列线程当然的前提是 `runningAsyncCalls` 线程数没有超上线，而且等待队列里面有等待的任务。

以上完成了线程线程池的复用和线程的管理工作。

小结，Call在执行任务通过Dispatcher把单元任务优先推到执行队列里进行操作，如果操作完成再执行等待队列的任务。

## 第十六节 Android高频面试专题 - 架构篇（二） okhttp面试必知必会

okhttp的火热程度，不用多说，已经被谷歌爸爸加入到Android源码中，也是面试高频的问题之一，如果只是满足于API工程师，那么面试还是有一点难度的。

### 1、HTTP报文结构

- 请求报文



- 响应报文



## 2、HTTP发展历史

- HTTP/0.9

只有一个命令GET

没有HEADER等描述数据的信息

服务器发送完毕，就关闭TCP连接

- HTTP/1.0

增加了很多命令

增加status code和header

多字符集支持、多部分发送、权限、缓存等

- HTTP/1.1

持久连接

pipeline

增加host和其他一些命令

- HTTP2

所有数据以二进制传输

同一个连接里面发送多个请求不再需要按照顺序来  
头信息压缩以及推送等提高效率的功能

**注意：目前使用最多的仍然是HTTP1.1，可以抓包看。**

### 3、okhttp有哪些优势

- 1)支持http2，对一台机器的所有请求共享同一个socket
- 2)内置连接池，支持连接复用，减少延迟
- 3)支持透明的gzip压缩响应体
- 4)通过缓存避免重复的请求
- 5)请求失败时自动重试主机的其他ip，自动重定向
- 6)丰富的API，可扩展性好

### 4、okhttp使用

```
//1. 创建OkHttpClient
OkHttpClient client = new OkHttpClient();
//2. 创建Request，并填入url信息
String url = "http://www.baidu.com";
Request request = new Request.Builder()
 .url(url)
 .build();
//3. 同步请求
Response response = client.newCall(request).execute();
//4. 异步请求
client.newCall(request).enqueue(responseCallback);
```

**\*\*5、看过okhttp源码吗？ \*\*简单介绍一下**

根据以上使用代码，不管同步还是异步请求，都是通过  
client.newCall(request)来进行执行，这个newCall其实是创建了一个  
RealCall对象，所以的请求处理，都是由RealCall来完成，RealCall在进行  
请求前，会坚持是否已经执行过，如果已执行会抛出异常，也就是说，一  
个Call对象只能处理一次请求。真正进行网络请求的是  
getResponseBodyWithInterceptorChain()方法，该方法内部将一系列的拦截  
器构成拦截链，然后链式执行proceed()方法完成网络请求。

### 6、同步请求详细源码解读

```
@Override public Response execute() throws IOException {
 synchronized (this) { //1.如果该请求正在运行抛出异常,
 //否则将运行标志位置为true, 防止重复请求
 if (executed) throw
 new IllegalStateException("Already Executed");
 executed = true; //2.捕获调用堆栈的跟踪
 captureCallStackTrace(); //告知eventlisten请求开始
 eventListener.callStart(this); try { //3.通过
 dispatcher.executed来实际执行
 client.dispatcher().executed(this); //4.经过一系列"拦截"
 //操作后获取结果
 Response result =
 getResponseWithInterceptorChain(); //如果result为空抛
 //出异常
 if (result == null) throw new
 IOException("Canceled");
 return result; } catch
 (IOException e) { //告知eventlisten请求失败
 eventListener.callFailed(this, e); throw e; }
 finally { //通知dispatcher执行完毕
 client.dispatcher().finished(this); }}
```

③中client.dispatcher().executed(this) 仅仅是将call加入同步请求队列，并没有真正开始进行网络请求，Dispatcher 内部维护着三个队列：同步请求队列 runningSyncCalls、异步请求队列 runningAsyncCalls、异步缓存队列 readyAsyncCalls，和一个线程池 executorService，来维护、管理、执行OKHttp的请求。④中getResponseWithInterceptorChain()才开始进行网络请求。

```
Response getResponsewithInterceptorChain() throws
IOException { //创建一个拦截器链表用于存放各种拦截器
List<Interceptor> interceptors = new ArrayList<>(); //
向链表中添加用户自定义的拦截器
interceptors.addAll(client.interceptors()); //1.向链表中
添加retryAndFollowUpInterceptor用于失败重试和重定向
interceptors.add(retryAndFollowUpInterceptor); //2.向链
表中添加BridgeInterceptor用于把用户构造的请求转换为发送给服务器的
请求，把服务器返回的响应转换为对用户友好的响应。
interceptors.add(new
BridgeInterceptor(client.cookieJar())); //3.向链表中添加
CacheInterceptor用于读取缓存以及更新缓存
interceptors.add(new
CacheInterceptor(client.internalCache())); //4.向链表中
添加ConnectInterceptor用于与服务器建立连接
interceptors.add(new ConnectInterceptor(client)); //如
果不是WebSocket添加networkInterceptors if
(!forWebSocket) {
interceptors.addAll(client.networkInterceptors()); }
//5.向链表中添加CallServerInterceptor用于从服务器读取响应的数据
interceptors.add(new
CallServerInterceptor(forWebSocket)); //根据上述的拦截器
链表生成一个拦截链 Interceptor.Chain chain = new
RealInterceptorChain(interceptors, null, null, null, 0,
originalRequest, this, eventListener,
client.connectTimeoutMillis(),
client.readTimeoutMillis(), client.writeTimeoutMillis());
//通过拦截链的proceed方法开始整个拦截链事件的传递 return
chain.proceed(originalRequest); }
```

主要列举一下默认已经实现的几个拦截器的作用：

RetryAndFollowUpInterceptor：重试和失败重定向拦截器

BridgeInterceptor：桥接拦截器，处理一些必须的请求头信息的拦截器

CacheInterceptor：缓存拦截器，用于处理缓存

ConnectInterceptor: 连接拦截器，建立可用的连接，是 CallServerInterceptor的基本

CallServerInterceptor: 请求服务器拦截器将 http 请求写进 IO 流当中，并且从 IO 流中读取响应 Response

具体细节，请阅读源码，这里不再进行细节描述。

<https://yq.aliyun.com/articles/78105>

## 7、异步请求详细源码解读

```
@Override public void enqueue(Callback responseCallback)
{ synchronized (this) { //1.如果该请求正在运行抛出异常，否则将运行标志位置为true，防止重复请求 if (executed) throw new
IllegalStateException("Already Executed"); executed =
true; } //2.捕获调用堆栈的跟踪 captureCallStackTrace();
eventListener.callStart(this); //3.通过dispatcher的
enqueue将此次请求添加到异步队列
client.dispatcher().enqueue(new
AsyncCall(responseCallback));}
```

enqueue实际上是new了一个RealCall的内部类AsyncCall扔进了 dispatcher中，如果当前正在运行的异步请求数小于阈值maxRequests（默认Dispatcher中为64）并且同host下运行的请求小于阈值maxRequestsPerHost（默认Dispatcher中为5），就将AsyncCall添加到正在运行的异步队里，并通过线程池异步执行，否则就将其丢到等待队列排队。

```
synchronized void enqueue(AsyncCall call) { if
(runningAsyncCalls.size() < maxRequests &&
runningCallsForHost(call) < maxRequestsPerHost) {
runningAsyncCalls.add(call);
executorService().execute(call); } else {
readyAsyncCalls.add(call); }}
```

AsyncCall实际上是一个Runnable，我们看一下进入线程池后真正执行的代码：

```
@Override protected void execute() { boolean
signalledCallback = false; try { //还是通过链式
调用实现 Response response =
getResponseWithInterceptorChain(); if
(retryAndFollowUpInterceptor.isCanceled()) {
signalledCallback = true;
responseCallback.onFailure(RealCall.this, new
IOException("Canceled")); } else {
signalledCallback = true;
responseCallback.onResponse(RealCall.this, response);
} } catch (IOException e) { if
(signalledCallback) { // Do not signal the
callback twice! Platform.get().log(INFO,
"Callback failure for " + toLoggableString(), e);
} else { eventListener.callFailed(RealCall.this,
e); responseCallback.onFailure(RealCall.this,
e); } } finally {
client.dispatcher().finished(this); } }
```

可以看到，内部还是跟同步请求一样，通过  
getResponseWithInterceptorChain()完成请求，然后通过传入的  
callBack回调请求结果，最后在finally中通知Dispatcher此次请求已完  
成，Dispatcher会在finish中检查当前请求数是否已低于阈值，若低于就  
去readyAsyncCalls等待队列中取出下一个请求。

## 8、okhttp实现网络请求的方法

OkHttp3的最底层是Socket，而不是URLConnection，它通过Platform的  
Class.forName()反射获得当前Runtime使用的socket库

socket发起网络请求的流程一般是：

- (1). 创建socket对象;
- (2). 连接到目标网络;
- (3). 进行输入输出流操作。

(1)(2)的实现，封装在connection接口中，具体的实现类是RealConnection。(3)是通过stream接口来实现，根据不同的网络协议，有Http1xStream和Http2xStream两个实现类，由于创建网络连接的时间较久(如果是HTTP的话，需要进行三次握手)，而请求经常是频繁的碎片化的，所以为了提高网络连接的效率，OKHttp3实现了网络连接复用。

## 9、okhttp实现带进度上传下载

OkHttp把请求和响应分别封装成了RequestBody和ResponseBody，下载进度自定义ResponseBody，重写source()方法，上传进度自定义RequestBody，重写writeTo()方法。

下载 <https://www.jianshu.com/p/df7d4945f007>

上传 <https://blog.csdn.net/u011247387/article/details/83027254>

## 10、为什么response.body().string() 只能调用一次

我们可能习惯在获取到Response对象后，先response.body().string()打印一遍log，再进行数据解析，却发现第二次直接抛异常，其实直接跟源码进去看就发现，通过source拿到字节流以后，直接给closeQuietly悄悄关闭了，这样第二次再去通过source读取就直接流已关闭的异常了。

```
public final String string() throws IOException {
 BufferedSource source = source(); try { Charset
 charset = Util.bomAwareCharset(source, charset());
 return source.readString(charset); } finally { //这里讲resource给悄悄close了 util.closeQuietly(source);
 } }
```

解决方案：1.内存缓存一份response.body().string(); 2.自定义拦截器处理log。

## 11、okhttp运用的设计模式

- 构造者模式 (OkhttpClient,Request等各种对象的创建)
- 工厂模式 (在Call接口中，有一个内部工厂Factory接口。 )
- 单例模式 (Platform类，已经使用Okhttp时使用单例)

- 策略模式（在CacheInterceptor中，在响应数据的选择中使用了策略模式，选择缓存数据还是选择网络访问。）
- 责任链模式（拦截器的链式调用）
- 享元模式（Dispatcher的线程池中，不限量的线程池实现了对象复用）
- ○ \*

## 第十七节 Android 网络优化，使用 HTTPDNS 优化 DNS，从原理到 OkHttp 集成

### 17.1 前言

谈到优化，首先第一步，肯定是把一个大功能，拆分成一个个细小的环节，再单个拎出来找到可以优化的点，App 的网络优化也是如此。

在 App 访问网络的时候，DNS 解析是网络请求的第一步，默认我们使用运营商的 LocalDNS 服务。有数据统计，在这一块 3G 网络下，耗时在 200~300ms，4G 网络下也需要 100ms。

解析慢，并不是 LocalDNS 最大的问题，它还存在一些更为严重的问题，例如：DNS 劫持、DNS 调度不准确（缓存、转发、NAT）导致性能退化等等，这些才是网络优化最应该解决的问题。

想要优化 DNS，现在最简单成熟的方案，就是使用 HTTPDNS。

今天就来聊聊，DNS、HTTPDNS，以及在 Android 下，如何使用 OKHttp 来集成 HTTPDNS。

### 17.2 DNS 和 HTTPDNS

#### 17.2.1 什么是 DNS

在说到 HTTPDNS 之前，先简单了解一下什么是 DNS？

在网络的世界中，每个有效的域名背后都有为其提供服务的服务器，而我们网络通信的首要条件，就是知道服务器的 IP 地址。

但是记住域名（网址）肯定是比记住 IP 地址简单。如果有某种方法，可以通过域名，查到其提供服务的服务器 IP 地址，那就非常方便了。这里就需要用到 DNS 服务器以及 DNS 解析。

**DNS (Domain Name System)**，它的作用就是根据域名，查出对应的 IP 地址，它是 HTTP 协议的前提。只有将域名正确的解析成 IP 地址后，后面的 HTTP 流程才可以继续进行下去。

DNS 服务器的要求，一定是高可用、高并发和分布式的服务器。它被分为多个层次结构。

- 根 DNS 服务器：返回顶级域 DNS 服务器的 IP 地址。
- 顶级域 DNS 服务器：返回权威 DNS 服务器的 IP 地址。
- 权威 DNS 服务器：返回相应主机的 IP 地址。

这三类 DNS 服务器，类似一种树状的结构，分级存在。

当开始 DNS 解析的时候，如果 LocalDNS 没有缓存，那就会向 LocalDNS 服务器请求（通常就是运营商），如果还是没有，就会一级一级的，从根域名查对应的顶级域名，再从顶级域名查权威域名服务器，最后通过权威域名服务器，获取具体域名对应的 IP 地址。

DNS 在提供域名和 IP 地址映射的过程中，其实提供了很多基于域名的功能，例如服务器的负载均衡，但是它也带来了一些问题。

### 17.2.2 DNS 的问题

DNS 的细节还有很多，本文就不展开细说了，其问题总结来说就是几点。

#### 1. 不稳定

DNS 劫持或者故障，导致服务不可用。

#### 2. 不准确

**LocalDNS 调度，并不一定是就近原则**，某些小运营商没有 DNS 服务器，直接调用其他运营商的 DNS 服务器，最终直接跨网传输。例如：用户侧是移动运营商，调度到了电信的 IP，造成访问慢，甚至访问受限等问题。

### 3. 不及时

运营商可能会修改 DNS 的 TTL(Time-To-Live，DNS 缓存时间)，导致 DNS 的修改，延迟生效。

还有运营商为了保证网内用户的访问质量，同时减少跨网结算，运营商会在网内搭建内容缓存服务器，通过把域名强行指向内容缓存服务器的地址，来实现本地本网流量完全留在本地的目的。

对此不同运营商甚至实现都不一致，这对我们来说就是个黑匣子。

正是因为 DNS 存在种种问题，所以牵出了 HTTPDNS。

#### 17.2.3 HTTPDNS 的解决方案

DNS 不仅支持 UDP，它还支持 TCP，但是大部分标准的 DNS 都是基于 UDP 与 DNS 服务器的 53 端口进行交互。

HTTPDNS 则不同，顾名思义它是利用 HTTP 协议与 DNS 服务器的 80 端口进行交互。不走传统的 DNS 解析，从而绕过运营商的 LocalDNS 服务器，有效的防止了域名劫持，提高域名解析的效率。

这就相当于，每家各自基于 HTTP 协议，自己实现了一套域名解析，自己去维护了一份域名与 IP 的地址簿，而不是使用同一的地址簿（DNS 服务器）。

据说微信有自己部署的 NETDNS，而各大云服务商，阿里云和腾讯云也提供了自己的 HTTPDNS 服务，对于我们普通开发者，只需要付出少量的费用，在手机端嵌入支持 HTTPDNS 的客户端 SDK，即可使用。

## 17.3 OkHttpClient 接入 HTTPDNS

既然了解了 HTTPDNS 的重要性，接下来看看如何在 OkHttpClient 中，集成 HTTPDNS。

OkHttpClient 是一个处理网络请求的开源项目，是 Android 端最火热的轻量级网络框架。在 OkHttpClient 中，默认是使用系统的 DNS 服务 InetAddress 进行域名解析。

```
IetAddress ip2=
IetAddress.getByName("www.cxmydev.com");
System.out.println(ip2.getHostAddress());
System.out.println(ip2.getHostName());
```

而想在 OkHttpClient 中使用 HTTPDNS，有两种方式。

1. 通过拦截器，在发送请求之前，将域名替换为 IP 地址。
2. 通过 OkHttpClient 提供的 `.dns()` 接口，配置 HTTPDNS。

对这两种方法来说，当然是推荐使用标准 API 来实现了。拦截器的方式，也建议有所了解，实现很简单，但是有坑。

### 17.3.1 拦截器接入方式

#### 1. 拦截器接入

拦截器是 OkHttpClient 中，非常强大的一种机制，它可以在请求和响应之间，做一些我们的定制操作。

在 OkHttpClient 中，可以通过实现 `Interceptor` 接口，来定制一个拦截器。使用时，只需要在 `OkHttpClient.Builder` 中，调用 `addInterceptor()` 方法来注册此拦截器即可。

OkHttpClient 的拦截器不是本文的重点，我们还是回到拦截器去实现 HTTPDNS 的话题上，拦截器没什么好说的，直接上相关代码。

```
class HTTPDNSInterceptor : Interceptor{
 override fun intercept(chain: Interceptor.Chain):
 Response {
```

```
val originRequest = chain.request()
val httpurl = originRequest.url()

val url = httpurl.toString()
val host = httpurl.host()

val hostIP = HttpDNS.getIpByHost(host)
val builder = originRequest.newBuilder()

if(hostIP!=null){
 builder.url(HttpDNS.getIpUrl(url,host,hostIP))
 builder.header("host",hostIP)
}
val newRequest = builder.build()
val newResponse = chain.proceed(newRequest)
return newResponse
}

}
```

在拦截器中，使用 `HttpDNS` 这个帮助类，通过 `getIpByHost()` 将 Host 转为对应的 IP。

如果通过抓包工具抓包，你会发现，原本的类似

`http://www.cxmydev.com/api/user` 的请求，被替换为：  
`http://220.181.57.xxx/api/user`。

## 2. 拦截器接入的坏处

使用拦截器，直接绕过了 DNS 的步骤，在请求发送前，将 Host 替换为对应的 IP 地址。

这种方案，在流程上很清晰，没有任何技术性的问题。但是这种方案存在一些问题，例如：HTTPS 下 IP 直连的证书问题、代理的问题、Cookie 的问题等等。

其中最严重的问题是，此方案（拦截器+HTTPDNS）遇到 https 时，如果存在一台服务器支持多个域名，可能导致证书无法匹配的问题。

在说到这个问题之前，就要先了解一下 HTTPS 和 SNI。

HTTPS 是为了保证安全的，在发送 HTTPS 请求之前，首先要进行 SSL/TLS 握手，握手的大致流程如下：

1. 客户端发起握手请求，携带随机数、支持算法列表等参数。
2. 服务端根据请求，选择合适的算法，下发公钥证书和随机数。
3. 客户端对服务端证书，进行校验，并发送随机数信息，该信息使用公钥加密。
4. 服务端通过私钥获取随机数信息。
5. 双方根据以上交互的信息，生成 Session Ticket，用作该连接后续数据传输的加密密钥。

在这个流程中，客户端需要验证服务器下发的证书。首先通过本地保存的根证书解开证书链，确认证书可信任，然后客户端还需要检查证书的 domain 域和扩展域，看看是否包含本次请求的 HOST。

在这一步就出现了问题，当使用拦截器时，请求的 URL 中，HOST 会被替换成 HTTPDNS 解析出来的 IP。当服务器存在多域名和证书的情况下，服务器在建立 SSL/TLS 握手时，无法区分到底应该返回那个证书，此时的策略可能返回默认证书或者不返回，这就有可能导致客户端在证书验证 domain 时，出现不匹配的情况，最终导致 SSL/TLS 握手失败。

这就引发出来 SNI 方案，SNI (Server Name Indication) 是为了解决一个服务器使用多个域名和证书的 SSL/TLS 扩展。

SNI 的工作原理，在连接到服务器建立 SSL 连接之前，先发送要访问站点的域名 (hostname)，服务器根据这个域名返回正确的证书。现在，大部分操作系统和浏览器，都已经很好的支持 SNI 扩展。

### 3. 拦截器 + HTTPDNS 的解决方案

这个问题，其实也有解决方案，这里简单介绍一下。

针对 "domain 不匹配" 的问题，可以通过 hook 证书验证过程中的第二步，将 IP 直接替换成原来的域名，再执行证书验证。

而 HttpURLConnection，提供了一个 HostnameVerifier 接口，实现它即可完成替换。

```
public interface HostnameVerifier { public boolean verify(String hostname, SSLSession session);}
```

如果使用 OkHttp，可以参考 OkHostnameVerifier  
(source://src/main/java/okhttp3/internal/tls/OkHostnameVerifier.java)  
的实现，进行替换。

本身 OkHttp 就不建议通过拦截器去做 HTTPDNS 的支持，所以这里就不展开讨论了，这里只提出解决的思路，有兴趣可以研究研究源码。

### 17.3.2 OKHttp 标准 API 接入

OkHttp 其实本身已经暴露了一个 Dns 接口，默认的实现是使用系统的 InetAddress 类，发送 UDP 请求进行 DNS 解析。

我们只需要实现 OkHttp 的 Dns 接口，即可获得 HTTPDNS 的支持。

在我们实现的 Dns 接口实现类中，解析 DNS 的方式，换成 HTTPDNS，将解析结果返回。

```
class HttpDns : Dns { override fun lookup(hostname: String): List<InetAddress> { val ip = HttpDnsHelper.getIpByHost(hostname) if (TextUtils.isEmpty(ip)) { // 返回自己解析的地址列表 return InetAddress.getAllByName(ip).toList() } else { // 解析失败，使用系统解析 return Dns.SYSTEM.lookup(hostname) } }}
```

使用也非常的简单，在 okhttp.build() 时，通过 dns() 方法配置。

```
mokHttpClient = httpBuilder .dns(HttpDns())
.build();
```

这样做的好处在于：

1. 还是用域名进行访问，只是底层 DNS 解析换成了 HTTPDNS，以确保解析的 IP 地址符合预期。
2. HTTPS 下的问题也得到解决，证书依然使用域名进行校验。

OkHttp 既然暴露出 dns 接口，我们就尽量使用它。

## 17.4小结时刻

现在大家知道，在做 App 的网络优化的时候，第一步就是使用 HTTPDNS 优化 DNS 的步骤。

所有的优化当然是以最终效果为目的，这里提两条大厂公开的数据，对腾讯的产品，在接入 HTTPDNS 后，用户平均延迟下降超过 10%，访问失败率下降超过五分之一。而百度 App 的 Feed 业务，Android 劫持率由 0.25% 降低到 0.05%。

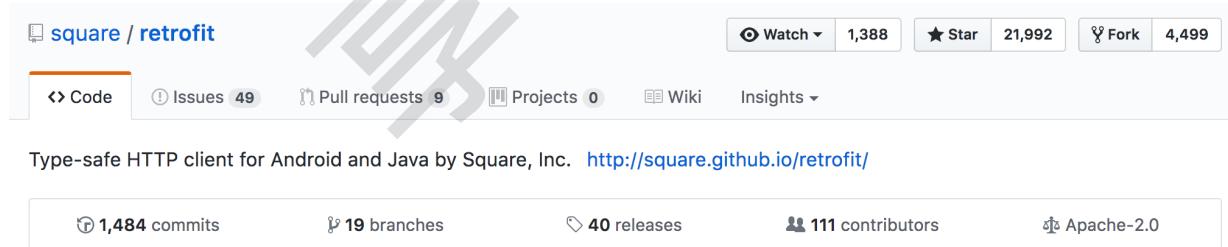
此种优化方案，非常依赖 HTTPDNS 服务器，所以建议使用 阿里云、腾讯云 这样相对稳定的云服务商。

## 第十八节 Retrofit源码分析

### 18.1 Android：手把手带你深入读懂Retrofit 2.0源码

#### 前言

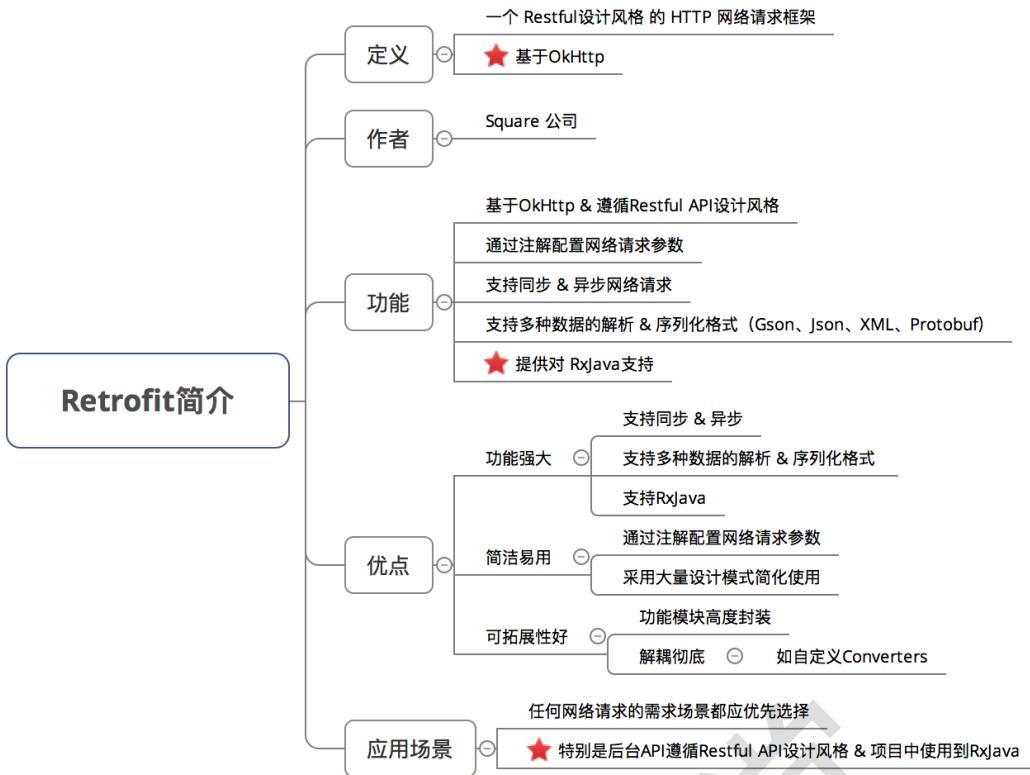
- 在Android开发中，网络请求十分常用
- 而在Android网络请求库中，Retrofit是当下最热的一个网络请求库



Github截图

- 今天，我将手把手带你深入剖析Retrofit v2.0的源码，希望你们会喜欢

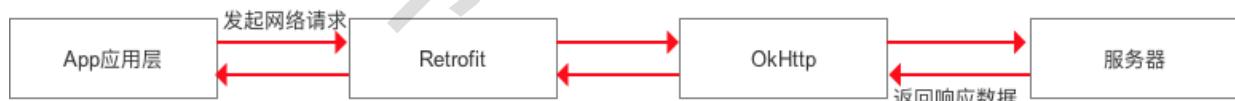
#### 1. 简介



## 示意图

特别注意：

- 准确来说，Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。
- 原因：网络请求的工作本质上是 OkHttp 完成，而 Retrofit 仅负责 网络请求接口的封装



## 流程图

- App 应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作
- 在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，Retrofit 根据用户的需求对结果进行解析
- \*

## 18.2 与其他网络请求开源库对比

除了Retrofit，如今Android中主流的网络请求框架有：

- Android-Async-Http
- Volley
- OkHttp

下面是简单介绍：

网络请求开源库 - 介绍

| 网络请求库 / 基础介绍          | android-async-http                              | Volley | OkHttp | Retrofit |
|-----------------------|-------------------------------------------------|--------|--------|----------|
| 作者                    | Loopj                                           | Google | Square | Square   |
| 面世时间                  | android-async-http > Volley > OkHttp > Retrofit |        |        |          |
| 人们使用情况 (GitHub Star数) | Volley > android-async-http > OkHttp > Retrofit |        |        |          |

网络请求加载 - 介绍

一图让你了解全部的网络请求库和他们之间的区别！

## 网络请求库- 对比

| 网络请求库 / 对比 | android-async-http                                                                                                                                                                                                                                                                                                     | Volley                                                                                                                                                                                                             | OkHttp                                                                                                                                                                                                                                                                                                                                                                            | Retrofit                                                                                                                                                                                                                                                                 |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 功能         | <ul style="list-style-type: none"> <li>• 基于HttpClient</li> <li>• 在 UI 线程外、异步进行Http请求</li> <li>• 在匿名回调中处理请求结果</li> <li>callback使用了Android的Handler发送消息机制在创建它的线程中执行           <ul style="list-style-type: none"> <li>• 自动智能请求重试</li> <li>• 持久化cookie存储</li> </ul> </li> <li>保存cookie到你的应用程序的 SharedPreferences</li> </ul> | <ul style="list-style-type: none"> <li>• 基于HttpURLConnection</li> <li>• 封装了UI图片加载框架，支持图片加载</li> <li>• 网络请求的排序、优先级处理</li> <li>• 缓存</li> <li>• 多级别取消请求</li> <li>• Activity和生命周期的联动（Activity结束时同时取消所有网络请求）</li> </ul> | <ul style="list-style-type: none"> <li>• 高性能Http请求库<br/>可把它理解成是一个封装之后的类似 HttpURLConnection 的一个东西，属于同级并不是基于上述二者；</li> <li>• 支持 SPDY，共享同一个Socket来处理同一个服务器的所有请求；</li> <li>• 支持http 2.0、websocket</li> <li>• 支持同步、异步</li> <li>• 封装了线程池、数据转换、参数使用、错误处理等</li> <li>• 无缝的支持GZIP来减少数据流量</li> <li>• 缓存响应数据来减少重复的网络请求</li> <li>• 能从很多常用的连接问题中自动恢复</li> <li>• 解决了代理服务器问题和SSL握手失败问题</li> </ul> | <ul style="list-style-type: none"> <li>• 基于OkHttp</li> <li>• RESTful Api设计风格</li> <li>• 支持同步、异步；</li> <li>• 通过注解配置请求</li> <li>包括请求方法，请求参数，请求头，返回值等</li> <li>• 可以搭配多种Converter将获得的数据解析&amp;序列化</li> <li>支持Gson（默认）、Jackson、Protobuf等</li> <li>• 提供对 RxJava 的支持</li> </ul> |
| 性能         |                                                                                                                                                                                                                                                                                                                        | <ul style="list-style-type: none"> <li>• 可拓展性好：可支持HttpClient、HttpURLConnection和OkHttp</li> </ul>                                                                                                                   | <ul style="list-style-type: none"> <li>• 基于 NIO 和 Okio， 所以性能更好：请求、处理速度快<br/>(IO：阻塞式；NIO：非阻塞式；Okio 是 Square 公司基于 IO 和 NIO 基础上做的一个更简单、高效处理数据流的一个库)</li> </ul>                                                                                                                                                                                                                       | <ul style="list-style-type: none"> <li>• 性能最好，处理最快；</li> <li>• 扩展性差<br/>高度封装所带来的必然后果；解析数据都是使用的统一的converter，如果服务器不能给出统一的API的形式，将很难进行处理。</li> </ul>                                                                                                                        |
| 开发者使用      | <p>1. 作者已经停止对该项目维护；<br/>2. Android5.0后不推荐用 HttpClient；<br/>所以不推荐在项目中使用了。</p>                                                                                                                                                                                                                                           | <ul style="list-style-type: none"> <li>• 封装性好：简单易用</li> </ul>                                                                                                                                                      | <ul style="list-style-type: none"> <li>• Api调用更加简单、方便；</li> <li>• 使用时需要进行多一层封装</li> </ul>                                                                                                                                                                                                                                                                                         | <ul style="list-style-type: none"> <li>• 简洁易用（Restful API设计风格）</li> <li>• 代码简化（更加高度的封装性和注解用法）</li> <li>• 解耦的更彻底、职责更细分</li> <li>• 易与其他框架联合使用（RxJava）</li> <li>• 使用方法较多，原理复杂，存在一定门槛</li> </ul>                                                                             |
| 应用场景       |                                                                                                                                                                                                                                                                                                                        | <ul style="list-style-type: none"> <li>• 适合轻量级网络交互：网络请求频繁、传输数据量小；</li> <li>• 不能进行大数据量的网络操作（比如下载视频、音频），所以不适合用来上传文件。</li> </ul>                                                                                      | <p>重量级网络交互场景：网络请求频繁、传输数据量大<br/>(其实会更推荐Retrofit，反正 Retrofit是基于Okhttp的)</p>                                                                                                                                                                                                                                                                                                         | <p>任何场景下优先选择，特别是：<br/>后台Api遵循RESTful的风格&amp;项目中有使用RxJava；</p>                                                                                                                                                                                                            |
| 备注         |                                                                                                                                                                                                                                                                                                                        | <p>Volley的request和response都是把数据放到byte数组里，不支持输入输出流，把数据放到数组中，如果大文件多了，数组就会非常的大且多，消耗内存。所以不如直接返回Stream那样具备可操作性，比如下载一个大文件，不可能把整个文件都缓存到内存之后再写到文件里。</p>                                                                    | <p>Android4.4的源码中可以看到<br/>HttpURLConnection已经替换成 OkHttp实现了。所以我们更有理由相信OkHttp的强大。</p>                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                          |

## 网络请求库 - 对比

### 附：各个主流网络请求库的Github地址

- [Android-Async-Http](#)
- [Volley](#)
- [OkHttp](#)
- [Retrofit](#)
- ○ \*

## 18.3 Retrofit 的具体使用

具体请看我写的文章：[这是一份很详细的 Retrofit 2.0 使用教程（含实例讲解）](#)

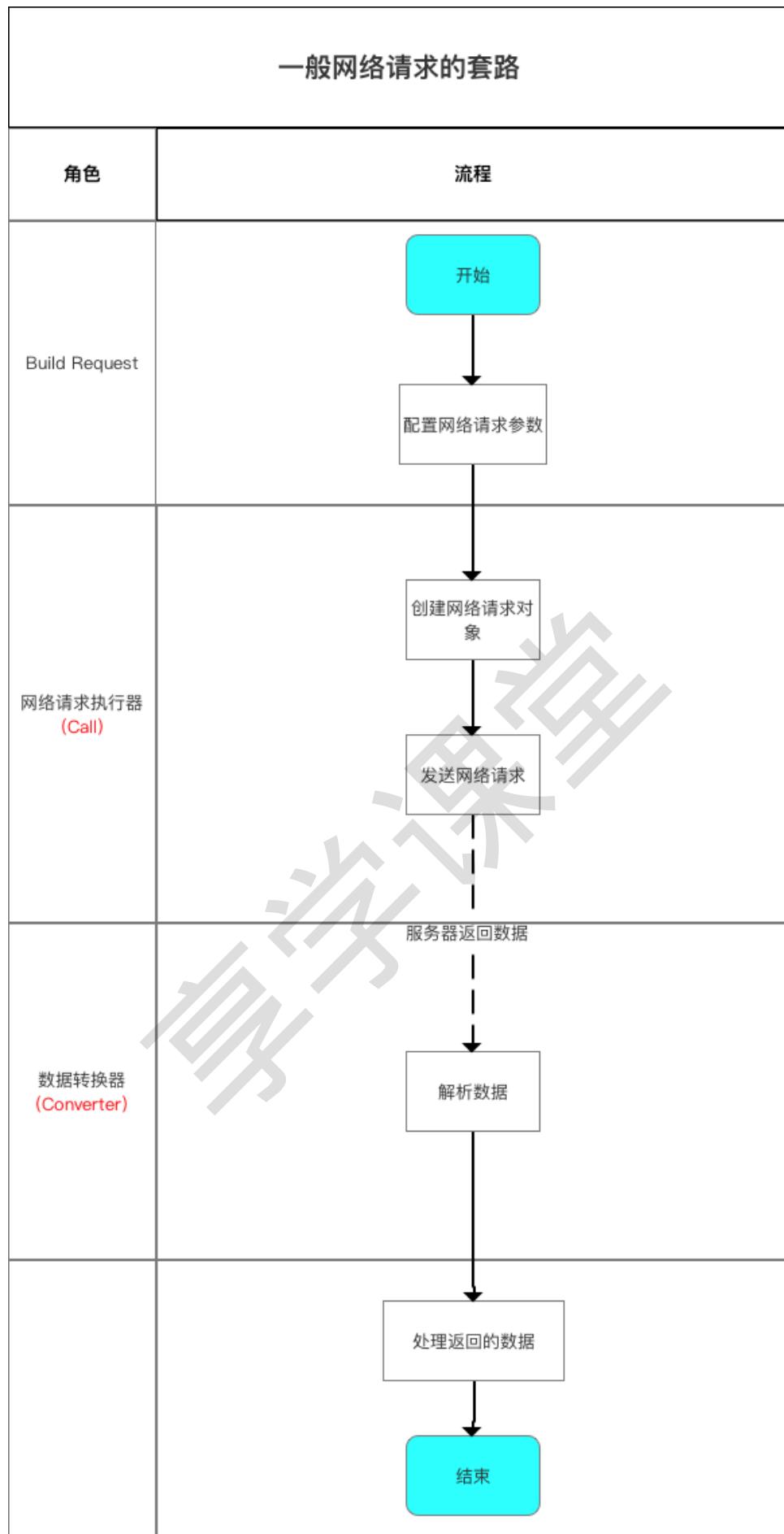
---

## 18.4 源码分析

### 18.4.1 Retrofit的本质流程

一般从网络通信过程如下图：



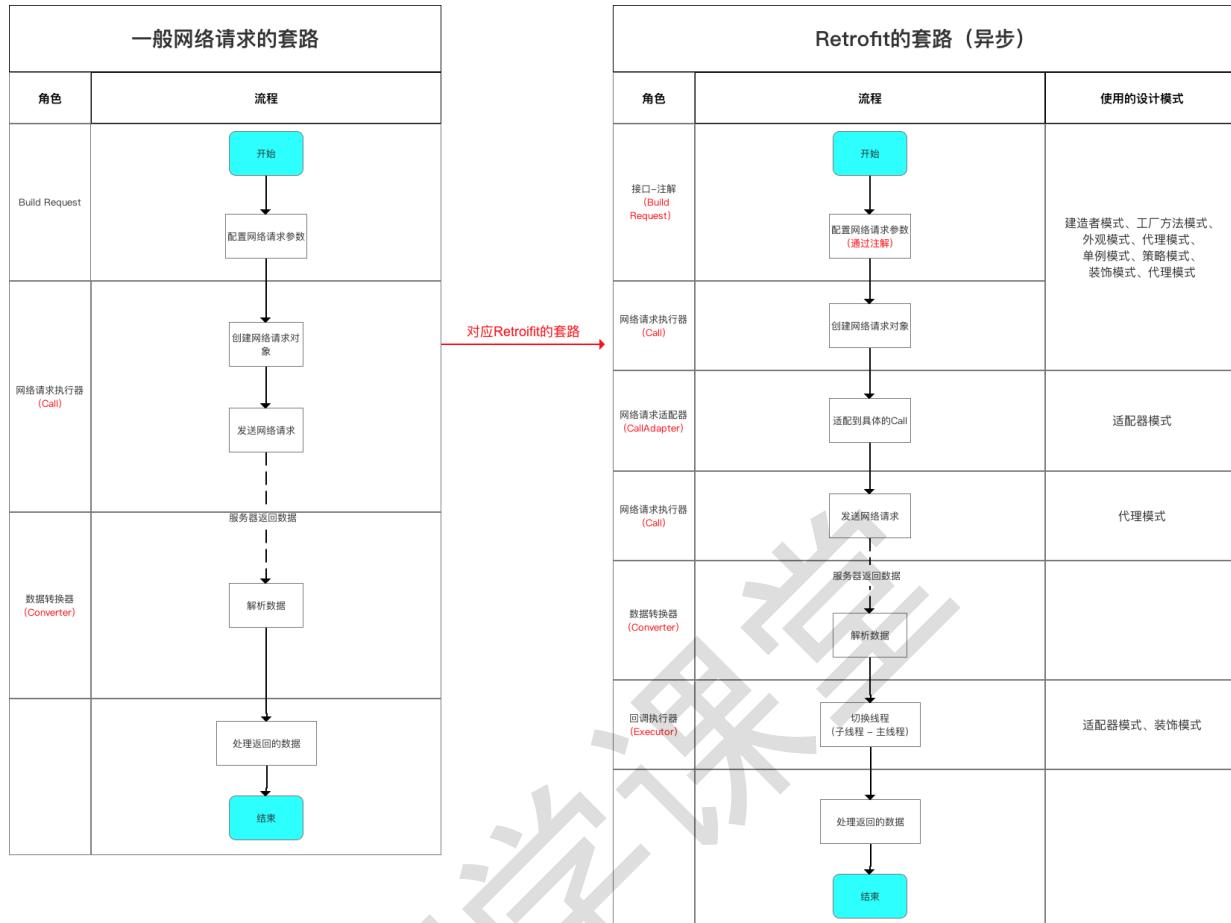


## 网络请求的过程

- 其实Retrofit的本质和上面是一样的套路

- 只是Retrofit通过使用大量的设计模式进行功能模块的解耦，使得上面的过程进行得更加简单 & 流畅

如下图：



Retrofit的本质

具体过程解释如下：

1. 通过解析 网络请求接口的注解 配置 网络请求参数
2. 通过 动态代理 生成 网络请求对象
3. 通过 网络请求适配器 将 网络请求对象 进行平台适配

平台包括：Android、Rxjava、Guava和java8

4. 通过 网络请求执行器 发送网络请求
5. 通过 数据转换器 解析服务器返回的数据
6. 通过 回调执行器 切换线程（子线程 ->>主线程）
7. 用户在主线程处理返回结果

下面介绍上面提到的几个角色

| 角色                          | 作用                                                                    | 备注                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 网络请求执行器<br>(Call)           | 创建 HTTP网络请求                                                           | Retrofit默认为 okhttp3.Call                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 网络请求适配器<br>(CallAdapter)    | 网络请求执行器 (Call) 的适配器，将默认的网络请求执行器 (OkHttpCall) 转换成立适合被不同平台来调用的网络请求执行器形式 | <ul style="list-style-type: none"> <li>Retrofit支持Android、RxJava、java8 和 Guava 四个平台；</li> <li>所以在Retrofit中提供了四种CallAdapterFactory：ExecutorCallAdapterFactory (Android默认)、GuavaCallAdapterFactory、Java8CallAdapterFactory、RxJavaCallAdapterFactory；</li> <li>网络适配器作用的理解：如：一开始Retrofit只打算利用OkHttpCall通过ExecutorCallbackCall切换线程；但后来发现使用Rxjava更加方便（不需要Handler来切换线程）。想要实现Rxjava的情况，那就得使用RxJavaCallAdapterFactory将OkHttpCall转换成Rxjava(Scheduler)；</li> <li>好处：用最小代价兼容更多平台，即能适配更多的使用场景</li> </ul> |
| 数据转换器<br>(Converter)        | 将返回数据解析成我们需要的数据类型                                                     | Retrofit支持如XML、Gson、JSON、protobuf等等                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 回调执行器<br>(CallBackExecutor) | 线程切换（子线程 -> 主线程）                                                      | 作用解释：将最后Okhttp的请求结果通过 callbackExecutor 使用Handler异步回调传回到主线程                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## 角色说明

**特别注意：因下面的 源码分析 是根据 使用步骤 逐步带你debug进去的，所以必须先看文章[这是一份很详细的 Retrofit 2.0 使用教程（含实例讲解）](#)**

### 18.4.2 源码分析

先来回忆Retrofit的使用步骤：

1. 创建Retrofit实例
2. 创建 网络请求接口实例 并 配置网络请求参数
3. 发送网络请求

封装了 数据转换、线程切换的操作

4. 处理服务器返回的数据

#### 4.2.1 创建Retrofit实例

##### a. 使用步骤

```

Retrofit retrofit = new Retrofit.Builder()
 .baseUrl("http://fanyi.youdao.com/")
 .addConverterFactory(GsonConverterFactory.create())
 .build();

```

## b. 源码分析

Retrofit实例是使用建造者模式通过Builder类进行创建的

建造者模式：将一个复杂对象的构建与表示分离，使得用户在不知道对象的创建细节情况下就可以直接创建复杂的对象。具体请看文章：  
[建造者模式 \(Builder Pattern\) - 最易懂的设计模式解析](#)

接下来，我将分五个步骤对创建Retrofit实例进行逐步分析

|                                                         |     |
|---------------------------------------------------------|-----|
| 步骤1                                                     | 步骤2 |
| Retrofit retrofit = new Retrofit.Builder()              |     |
| .baseUrl("http://fanyi.youdao.com/") 步骤3                |     |
| .addConverterFactory(GsonConverterFactory.create()) 步骤4 |     |
| .build(); 步骤5                                           |     |

分析步骤

步骤1

|                                                         |     |
|---------------------------------------------------------|-----|
| 步骤1                                                     | 步骤2 |
| Retrofit retrofit = new Retrofit.Builder()              |     |
| .baseUrl("http://fanyi.youdao.com/") 步骤3                |     |
| .addConverterFactory(GsonConverterFactory.create()) 步骤4 |     |
| .build(); 步骤5                                           |     |

步骤1

```
<-- Retrofit类 --> public final class Retrofit {
 private final Map<Method, ServiceMethod>
 serviceMethodCache = new LinkedHashMap<>(); // 网络请求配
置对象（对网络请求接口中方法注解进行解析后得到的对象） // 作用：存
储网络请求相关的配置，如网络请求的方法、数据转换器、网络请求适配器、
网络请求工厂、基地址等 private final HttpUrl baseUrl; //
网络请求的url地址 private final okhttp3.Call.Factory
callFactory; // 网络请求器的工厂 // 作用：生产网络请求器
(Call) // Retrofit是默认使用okhttp private final
List<CallAdapter.Factory> adapterFactories; // 网络请求适
配器工厂的集合 // 作用：放置网络请求适配器工厂 // 网络请求适配器
工厂作用：生产网络请求适配器 (CallAdapter) // 下面会详细说明
private final List<Converter.Factory> converterFactories;
// 数据转换器工厂的集合 // 作用：放置数据转换器工厂 // 数据转换
器工厂作用：生产数据转换器 (converter) private final
Executor callbackExecutor; // 回调方法执行器 private final
boolean validateEagerly; // 标志位 // 作用：是否提前对业务接口中
的注解进行验证转换的标志位<-- Retrofit类的构造函数 -->
>Retrofit(okhttp3.Call.Factory callFactory, HttpUrl
baseUrl, List<Converter.Factory>
converterFactories, List<CallAdapter.Factory>
adapterFactories, Executor callbackExecutor,
boolean validateEagerly) { this.callFactory =
callFactory; this.baseUrl = baseUrl;
this.converterFactories =
unmodifiableList(converterFactories);
this.adapterFactories =
unmodifiableList(adapterFactories); //
unmodifiableList(list)近似于UnmodifiableList<E>(list)
// 作用：创建的新对象能够对list数据进行访问，但不可通过该对象对list
集合中的元素进行修改 this.callbackExecutor =
callbackExecutor; this.validateEagerly =
validateEagerly; ... // 仅贴出关键代码}
```

**成功建立一个Retrofit对象的标准：配置好Retrofit类里的成员变量，即  
配置好：**

- `serviceMethod`: 包含所有网络请求信息的对象

- `baseUrl`: 网络请求的url地址
- `callFactory`: 网络请求工厂
- `adapterFactories`: 网络请求适配器工厂的集合
- `converterFactories`: 数据转换器工厂的集合
- `callbackExecutor`: 回调方法执行器

所谓`xxxFactory`、“`xxx工厂`”其实是设计模式中**工厂模式**的体现：将“类实例化的操作”与“使用对象的操作”分开，使得使用者不用知道具体参数就可以实例化出所需要的“产品”类。

这里详细介绍一下：`CallAdapterFactory`：该`Factory`生产的是`CallAdapter`，那么`CallAdapter`又是什么呢？

### `CallAdapter`详细介绍

- 定义：网络请求执行器（Call）的适配器
  1. Call在Retrofit里默认是`okHttpCall`
  2. 在Retrofit中提供了四种`CallAdapterFactory`：  
`ExecutorCallAdapterFactory`（默认）、  
`GuavaCallAdapterFactory`、`Java8CallAdapterFactory`、  
`RxJavaCallAdapterFactory`
- 作用：将默认的网络请求执行器（`OkHttpCall`）转换成适合被不同平台来调用的网络请求执行器形式

1. 如：一开始Retrofit只打算利用`okHttpCall`通过`ExecutorCallbackCall`切换线程；但后来发现使用`Rxjava`更加方便（不需要Handler来切换线程）。想要实现`Rxjava`的情况，那就得使用`RxJavaCallAdapterFactory``CallAdapter`将`okHttpCall`转换成`Rxjava(Scheduler)`：

```
// 把response封装成rxjava的Observable，然后进行流式操作
Retrofit.Builder.addCallAdapterFactory(new RxJavaCallAdapterFactory().create()); // 关于RxJava的使用这里不作更多的展开
```

2. Retrofit还支持java8、Guava平台。

- 好处：用最小代价兼容更多平台，即能适配更多的使用场景

所以，接下来需要分析的步骤2、步骤3、步骤4、步骤4的目的是配置好上述所有成员变量

## 步骤2

```
步骤1 步骤2
Retrofit retrofit = new Retrofit.Builder()
.baseUrl("http://fanyi.youdao.com/") 步骤3
.addConverterFactory(GsonConverterFactory.create()) 步骤4
.build(); 步骤5
```

## 步骤2

我们先来看Builder类

| 请按下面提示的步骤进行查看

想  
識  
思  
學

```
<-- Builder类-->public static final class Builder {
 private Platform platform; private
 okhttp3.Call.Factory callFactory; private HttpUrl
 baseUrl; private List<Converter.Factory>
 converterFactories = new ArrayList<>(); private
 List<CallAdapter.Factory> adapterFactories = new
 ArrayList<>(); private Executor callbackExecutor;
 private boolean validateEagerly;// 从上面可以发现， Builder
 类的成员变量与Retrofit类的成员变量是对应的// 所以Retrofit类的成员
 变量基本上是通过Builder类进行配置// 开始看步骤1<-- 步骤1 -->/
 Builder的构造方法（无参） public Builder() {
 this(Platform.get());// 用this调用自己的有参构造方法public
 Builder(Platform platform) ->>步骤5（看完步骤2、3、4再看） //
 并通过调用Platform.get () 传入了Platform对象// 继续看
 Platform.get()方法 ->>步骤2// 记得最后继续看步骤5的Builder有参
 构造方法 }...}<-- 步骤2 -->class Platform { private
 static final Platform PLATFORM = findPlatform(); // 将
 findPlatform()赋给静态变量 static Platform get() {
 return PLATFORM; // 返回静态变量PLATFORM, 即
 findPlatform() ->>步骤3 }<-- 步骤3 -->private static
 Platform findPlatform() { try {
 Class.forName("android.os.Build"); //
 Class.forName(xxx.xx.xx)的作用：要求JVM查找并加载指定的类（即
 JVM会执行该类的静态代码段） if (Build.VERSION.SDK_INT !=
 0) { return new Android(); // 此处表示：如果
 是Android平台，就创建并返回一个Android对象 ->>步骤4 } }
 catch (ClassNotFoundException ignored) { } try {
 // 支持Java平台
 Class.forName("java.util.Optional"); return new
 Java8(); } catch (ClassNotFoundException ignored) {
 } try { // 支持ios平台
 Class.forName("org.robovm.apple.foundation.NSObject");
 return new IOS(); } catch (ClassNotFoundException
 ignored) { } // 从上面看出：Retrofit2.0支持3个平台：Android
 平台、Java平台、IOS平台// 最后返回一个Platform对象（指定了
 Android平台）给Builder的有参构造方法public Builder(Platform
 platform) -> 步骤5// 说明Builder指定了运行平台为Android
 return new Platform(); }...}<-- 步骤4 -->/> // 用于接收服务器返
```

```
回数据后进行线程切换在主线程显示结果static class Android
extends Platform { @Override CallAdapter.Factory
defaultCallAdapterFactory(Executor callbackExecutor) {
 return new
ExecutorCallAdapterFactory(callbackExecutor); // 创建默
认的网络请求适配器工厂 // 该默认工厂生产的 adapter 会使得call
在异步调用时在指定的 Executor 上执行回调 // 在Retrofit中提供
了四种CallAdapterFactory: ExecutorCallAdapterFactory (默
认)、GuavaCallAdapterFactory、Java8CallAdapterFactory、
RxJavaCallAdapterFactory // 采用了策略模式 }
@Override public Executor defaultCallbackExecutor()
{ // 返回一个默认的回调方法执行器 // 该执行器作用: 切换
线程 (子->>主线程), 并在主线程 (UI线程) 中执行回调方法
return new MainThreadExecutor(); } static class
MainThreadExecutor implements Executor { private
final Handler handler = new
Handler(Looper.getMainLooper()); // 获取与Android 主线
程绑定的Handler @Override public void
execute(Runnable r) {
handler.post(r); // 该Handler是上面获取的与Android 主
线程绑定的Handler // 在UI线程进行对网络请求返回数据处理等
操作。 } } // 切换线程的流程: // 1. 回调
ExecutorCallAdapterFactory生成了一个ExecutorCallbackCall对
象//2. 通过调用ExecutorCallbackCall.enqueue(CallBack)从而调
用MainThreadExecutor的execute()通过handler切换到主线程 }//
下面继续看步骤5的Builder有参构造方法<-- 步骤5 -->// Builder类
的构造函数2 (有参) public Builder(Platform platform) {
// 接收Platform对象 (Android平台) this.platform =
platform; // 通过传入BuiltInConverters()对象配置数据转换器工
厂 (converterFactories) // converterFactories是一个存放数据转
换器Converter.Factory的数组// 配置converterFactories即配置里
面的数据转换器 converterFactories.add(new
BuiltInConverters()); // BuiltInConverters是一个内置的数据转
换器工厂 (继承Converter.Factory类) // new BuiltInConverters()
是为了初始化数据转换器 }
```

对Builder类分析完毕，总结：Builder设置了默认的

- 平台类型对象：Android
- 网络请求适配器工厂：CallAdapterFactory

CallAdapter用于对原始Call进行再次封装，如Call到Observable

- 数据转换器工厂：ConverterFactory
- 回调执行器：callbackExecutor

**特别注意，这里只是设置了默认值，但未真正配置到具体的Retrofit类的成员变量当中**

### 步骤3



### 步骤3

还是按部就班按步骤来观看

```
<-- 步骤1 -->public Builder baseUrl(String baseUrl) {
 // 把String类型的url参数转化为适合OkHttp的HttpUrl类型
 HttpUrl httpUrl = HttpUrl.parse(baseUrl); // 最终
 返回带HttpUrl类型参数的baseUrl() // 下面继续看
 baseUrl(httpUrl) ->> 步骤2 return baseUrl(httpUrl);
} <-- 步骤2 --> public Builder baseUrl(HttpUrl baseUrl)
{ // 把URL参数分割成几个路径碎片 List<String>
 pathSegments = baseUrl.pathSegments(); // 检测最后
 一个片段来检查URL参数是不是以"/"结尾 // 不是就抛出异常
 if (!"".equals(pathSegments.get(pathSegments.size() -
 1))) { throw new IllegalArgumentException("baseUrl
 must end in /: " + baseUrl); }
 this.baseUrl = baseUrl; return this; }
```

- 至此，步骤3分析完毕
- 总结：**baseUrl () 用于配置Retrofit类的网络请求url地址**

将传入的String类型url转化为适合OkHttp的HttpUrl类型的url

## 步骤4

```
步骤1 步骤2
Retrofit retrofit = new Retrofit.Builder()
.baseUrl("http://fanyi.youdao.com/") 步骤3
.addConverterFactory(GsonConverterFactory.create()) 步骤4
.build(); 步骤5
```

## 步骤4

我们从里往外看，即先看 `GsonConverterFactory.create()`

```
public final class GsonConverterFactory extends
Converter.Factory {<-- 步骤1 --> public static
GsonConverterFactory create() { // 创建一个Gson对象
return create(new Gson()); -> 步骤2 }<-- 步骤2 --> public
static GsonConverterFactory create(Gson gson) { // 创建
了一个含有Gson对象实例的GsonConverterFactory return new
GsonConverterFactory(gson); -> 步骤3 } private final
Gson gson;<-- 步骤3 --> private GsonConverterFactory(Gson
gson) { if (gson == null) throw new
NullPointerException("gson == null"); this.gson =
gson; }
```

- 所以，`GsonConverterFactory.create()`是创建了一个含有Gson对象实例的`GsonConverterFactory`，并返回给`addConverterFactory()`
- 接下来继续看：`addConverterFactory()`

```
// 将上面创建的GsonConverterFactory放入到 converterFactories
数组// 在第二步放入一个内置的数据转换器工厂BuiltInConverters()后
又放入了一个GsonConverterFactory public Builder
addConverterFactory(Converter.Factory factory) {
converterFactories.add(checkNotNull(factory, "factory ==
null")); return this; }
```

- 至此，分析完毕
- 总结：步骤4用于创建一个含有Gson对象实例的`GsonConverterFactory`并放入到数据转换器工厂`converterFactories`里

1. 即Retrofit默认使用Gson进行解析
2. 若使用其他解析方式（如json、XML或Protobuf），也可通过自定义数据解析器来实现（必须继承 Converter.Factory）

## 步骤5

```
步骤1 步骤2
Retrofit retrofit = new Retrofit.Builder()
 .baseUrl("http://fanyi.youdao.com/") 步骤3
 .addConverterFactory(GsonConverterFactory.create()) 步骤4
 .build(); 步骤5
```

## 步骤5

终于到了最后一个步骤了。

第十一课 基于Retrofit的Android网络请求

```
public Retrofit build() { <-- 配置网络请求执行器
(callFactory) --> okhttp3.Call.Factory callFactory =
this.callFactory; // 如果没指定，则默认使用okhttp
// 所以Retrofit默认使用okhttp进行网络请求 if
(callFactory == null) { callFactory = new
OkHttpClient(); } <-- 配置回调方法执行器
(callbackExecutor) --> Executor callbackExecutor =
this.callbackExecutor; // 如果没指定，则默认使用Platform
检测环境时的默认callbackExecutor // 即Android默认的
callbackExecutor if (callbackExecutor == null) {
callbackExecutor = platform.defaultCallbackExecutor();
} <-- 配置网络请求适配器工厂 (CallAdapterFactory) -->
List<CallAdapter.Factory> adapterFactories = new
ArrayList<>(this.adapterFactories); // 向该集合中添加了
步骤2中创建的CallAdapter.Factory请求适配器 (添加在集合器末尾)

adapterFactories.add(platform.defaultCallAdapterFactory(c
allbackExecutor)); // 请求适配器工厂集合存储顺序：自定义1适配
器工厂、自定义2适配器工厂...默认适配器工厂
(ExecutorCallAdapterFactory) <-- 配置数据转换器工厂：
converterFactory --> // 在步骤2中已经添加了内置的数据转换
器BuiltInConverters() (添加到集合器的首位) // 在步骤4中
又插入了一个Gson的转换器 - GsonConverterFactory (添加到集合器的
首二位) List<Converter.Factory> converterFactories =
new ArrayList<>(this.converterFactories); // 数据转换
器工厂集合存储的是：默认数据转换器工厂 (BuiltInConverters) 、自
定义1数据转换器工厂 (GsonConverterFactory) 、自定义2数据转换器工
厂....// 注： //1. 获取合适的网络请求适配器和数据转换器都是从
adapterFactories和converterFactories集合的首位-末位开始遍历//
因此集合中的工厂位置越靠前就拥有越高的使用权 // 最终返回一个
Retrofit的对象，并传入上述已经配置好的成员变量 return new
Retrofit(callFactory, baseUrl, converterFactories,
adapterFactories, callbackExecutor,
validateEagerly); }
```

- 至此，步骤5分析完毕

- 总结：在最后一步中，通过前面步骤设置的变量，将Retrofit类的所有成员变量都配置完毕。
- 所以，成功创建了Retrofit的实例

## 总结

Retrofit使用建造者模式通过Builder类建立了一个Retrofit实例，具体创建细节是配置了：

- 平台类型对象 (Platform - Android)
- 网络请求的url地址 (baseUrl)
- 网络请求工厂 (callFactory)

默认使用OkHttpCall

- 网络请求适配器工厂的集合 (adapterFactories)

本质是配置了网络请求适配器工厂 - 默认是 ExecutorCallAdapterFactory

- 数据转换器工厂的集合 (converterFactories)

本质是配置了数据转换器工厂

- 回调方法执行器 (callbackExecutor)

默认回调方法执行器作用是：切换线程（子线程 - 主线程）

由于使用了建造者模式，所以开发者并不需要关心配置细节就可以创建好 Retrofit实例，建造者模式get。

在创建Retrofit对象时，你可以通过更多更灵活的方式去处理你的需求，如使用不同的Converter、使用不同的CallAdapter，这也就提供了你使用RxJava来调用Retrofit的可能

## 2. 创建网络请求接口的实例

### 2.1 使用步骤

```

<-- 步骤1: 定义接收网络数据的类 --><-- JavaBean.java -->public
class JavaBean { .. // 这里就不介绍了 }<-- 步骤2: 定义网络请
求的接口类 --><-- AccessApi.java -->public interface
AccessApi { // 注解GET: 采用Get方法发送网络请求 //
Retrofit把网络请求的URL分成了2部分: 1部分baseUrl放在创建
Retrofit对象时设置; 另一部分在网络请求接口设置(即这里) // 如
果接口里的URL是一个完整的网址, 那么放在创建Retrofit对象时设置的部分
可以不设置 @GET("openapi.do?
keyfrom=Yanzhikai&key=2032414398&type=data&doctype=json&
version=1.1&q=car") // 接受网络请求数据的方法
Call<JavaBean> getCall(); // 返回类型为Call<*>, *是解析得
到的数据类型, 即JavaBean}<-- 步骤3: 在MainActivity创建接口类实例
-->AccessApi NetService =
retrofit.create(AccessApi.class); <-- 步骤4: 对发送请
求的url进行封装, 即生成最终的网络请求对象 -->
Call<JavaBean> call = NetService.getCall();

```

## 2.2 源码分析

- 结论: Retrofit是通过外观模式 & 代理模式 使用create () 方法创建网络请求接口的实例 (同时, 通过网络请求接口里设置的注解进行了网络请求参数的配置)
- 2. 外观模式: 定义一个统一接口, 外部与通过该统一的接口对子系统里的其他接口进行访问。具体请看: [外观模式 \(Facade Pattern\) - 最易懂的设计模式解析](#)
- 3. 代理模式: 通过访问代理对象的方式来间接访问目标对象。具体请看: [代理模式 \(Proxy Pattern\) - 最易懂的设计模式解析](#)
- 下面主要分析步骤3和步骤4:

```

<-- 步骤3: 在MainActivity创建接口类实例 -->AccessApi
NetService = retrofit.create(NetService.class);<-- 步骤4:
对发送请求的url进行封装, 即生成最终的网络请求对象 -->
Call<JavaBean> call = NetService.getCall();

```

\*\*步骤3讲解: AccessApi NetService =  
 retrofit.create(Ne\*\*tService.class);

想  
識  
思  
學

```
public <T> T create(final Class<T> service) { if
(validateEagerly) { // 判断是否需要提前验证
eagerlyValidateMethods(service); // 具体方法作用:
// 1. 给接口中每个方法的注解进行解析并得到一个ServiceMethod对象
// 2. 以Method为键将该对象存入LinkedHashMap集合中 // 特
别注意: 如果不是提前验证则进行动态解析对应方法(下面会详细说明), 得
到一个ServiceMethod对象, 最后存入到LinkedHashMap集合中, 类似延迟
加载(默认) } // 创建了网络请求接口的动态代理对象,
即通过动态代理创建网络请求接口的实例(并最终返回) // 该动
态代理是为了拿到网络请求接口实例上所有注解 return (T)
Proxy.newProxyInstance(
service.getClassLoader(), // 动态生成接口的实现类
new Class<?>[] { service }, // 动态创建实例
new InvocationHandler() { // 将代理类的实现交给
InvocationHandler类作为具体的实现(下面会解释)
private final Platform platform = Platform.get();
// 在 InvocationHandler类的invoke()实现中, 除了执行真正的逻辑
// 如再次转发给真正的实现类对象, 还可以进行一些有用的操作
// 如统计执行时间、进行初始化和清理、对接口调用进行检查等。
@Override public Object invoke(Object proxy,
Method method, Object... args) throws
Throwable { // 下面会详细介绍 invoke
() 的实现 // 即下面三行代码
ServiceMethod serviceMethod = loadServiceMethod(method);
OkHttpCall okHttpCall = new OkHttpCall<>
(serviceMethod, args); return
serviceMethod.callAdapter.adapt(okHttpCall); }
}); } // 特别注意// return (T)
Proxy.newProxyInstance(ClassLoader loader, Class<?>[]
interfaces, InvocationHandler invocationHandler)// 可以解
读为: getProxyClass(loader, interfaces)
.getConstructor(InvocationHandler.class).newInstance(invoca
tionHandler); // 即通过动态生成的代理类, 调用interfaces接口的
方法实际上是通过调用InvocationHandler对象的invoke()来完成指定
的功能// 先记住结论, 在讲解步骤4的时候会再次详细说明<-- 关注点1:
eagerlyValidateMethods() -->private void
eagerlyValidateMethods(Class<?> service) { Platform
platform = Platform.get(); for (Method method :
```

```
service.getDeclaredMethods()) { if
(!platform.isDefaultMethod(method)) {
loadServiceMethod(method); } // 将传入的ServiceMethod
对象加入LinkedHashMap<Method, ServiceMethod>集合 // 使用
LinkedHashMap集合的好处:
lruEntries.values().iterator().next()获取到的是集合最不经常用
到的元素，提供了一种Lru算法的实现 } }
```

创建网络接口实例用了外观模式 & 代理模式：

使用外观模式进行访问，里面用了代理模式

## 1. 外观模式

- 外观模式：定义一个统一接口，外部与通过该统一的接口对子系统里的其他接口进行访问。具体请看：[外观模式 \(Facade Pattern\) - 最易懂的设计模式解析](#)
- Retrofit对象的外观（门店） = `retrofit.create()`
- 通过这一外观方法就可以在内部调用各个方法创建网络请求接口的实例和配置网络请求参数

大大降低了系统的耦合度

## 2. 代理模式

- 代理模式：通过访问代理对象的方式来间接访问目标对象

分为静态代理 & 动态代理：

1. 静态代理：代理类在程序运行前已经存在的代理方式
2. 动态代理：代理类在程序运行前不存在、运行时由程序动态生成的代理方式

具体请看文章[代理模式 \(Proxy Pattern\) - 最易懂的设计模式解析](#)

- `return (T) roxy.newProxyInstance(classLoader loader,
class<?>[] interfaces, InvocationHandler invocationHandler)` 通过代理模式中的动态代理模式，动态生成网络请求接口的代理类，并将代理类的实例创建交给 `InvocationHandler` 类作为具体的实现，并最终返回一个动态代理对象。

生成实例过程中含有生成实现类的缓存机制（单例模式），下面会详细分析

使用动态代理的好处：

- 当NetService对象调用getCall()接口中方法时会进行拦截，调用都会集中转发到InvocationHandler#invoke()，可集中进行处理
- 获得网络请求接口实例上的所有注解
- 更方便封装ServiceMethod

## 下面看源码分析

下面将详细分析InvocationHandler类 # invoke() 里的具体实现

```
new InvocationHandler() { private final
Platform platform = Platform.get(); @Override
public Object invoke(Object proxy, Method method,
Object... args) throws Throwable {
// 将详细介绍下面代码 // 关注点1 // 作
用：读取网络请求接口里的方法，并根据前面配置好的属性配置
serviceMethod对象 serviceMethod serviceMethod =
loadServiceMethod(method); //
关注点2 // 作用：根据配置好的serviceMethod对象创建
okHttpCall对象 okHttpCall okHttpCall = new
okHttpCall<>(serviceMethod, args); // 关注点3
 // 作用：调用okHttp，并根据okHttpCall返回rejava的
Observe对象或者返回Call return
serviceMethod.callAdapter.adapt(okHttpCall); }
```

下面将详细介绍3个关注点的代码。

**关注点1：** `serviceMethod serviceMethod =`  
`loadServiceMethod(method);`

```

<-- loadServiceMethod(method)方法讲解 --> // 一个
ServiceMethod 对象对应于网络请求接口里的一个方法 //
loadServiceMethod (method) 负责加载 ServiceMethod:
ServiceMethod loadServiceMethod(Method method) {
ServiceMethod result; // 设置线程同步锁 synchronized
(serviceMethodCache) { result =
serviceMethodCache.get(method); // ServiceMethod类对
象采用了单例模式进行创建 // 即创建ServiceMethod对象前，先看
serviceMethodCache有没有缓存之前创建过的网络请求实例
// 若没缓存，则通过建造者模式创建 serviceMethod 对象 if
(result == null) { // 下面会详细介绍ServiceMethod生成实
例的过程 result = new ServiceMethod.Builder(this,
method).build(); serviceMethodCache.put(method,
result); } } return result; }// 这里就是上面说的
创建实例的缓存机制：采用单例模式从而实现一个 ServiceMethod 对象对
应于网络请求接口里的一个方法// 注：由于每次获取接口实例都是传入
class 对象// 而 class 对象在进程内单例的，所以获取到它的同一个方法
Method 实例也是单例的，所以这里的缓存是有效的。

```

下面，我将分3个步骤详细分析 serviceMethod 实例的创建过程：

步骤1  
result = new ServiceMethod.Builder(this, method).  
步骤2  
步骤3  
.build();

Paste\_Image.png

### 步骤1： ServiceMethod类 构造函数

步骤1  
result = new ServiceMethod.Builder(this, method).  
步骤2  
步骤3  
.build();

Paste\_Image.png

```
<-- ServiceMethod 类 -->public final class ServiceMethod
{final okhttp3.Call.Factory callFactory; // 网络请求工厂
final CallAdapter<?> callAdapter; // 网络请求适配器工厂// 具体创建是在new ServiceMethod.Builder(this, method).build()最后的build()中// 下面会详细说明private final
Converter<ResponseBody, T> responseConverter; // Response
内容转换器 // 作用: 负责把服务器返回的数据 (JSON或者其他格式, 由
ResponseBody 封装) 转化为 T 类型的对象; private final
HttpUrl baseUrl; // 网络请求地址 private final String
relativeUrl; // 网络请求的相对地址 private final String
httpMethod; // 网络请求的Http方法 private final Headers
headers; // 网络请求的http请求头 键值对 private final
MediaType contentType; // 网络请求的http报文body的类型
private final ParameterHandler<?>[] parameterHandlers;
// 方法参数处理器 // 作用: 负责解析 API 定义时每个方法的参数, 并在构造 HTTP 请求时设置参数; // 下面会详细说明// 说明: 从上面的成员变量可以看出, ServiceMethod对象包含了访问网络的所有基本信息<-->
ServiceMethod 类的构造函数 -->// 作用: 传入各种网络请求参数
ServiceMethod(Builder<T> builder) { this.callFactory =
builder.retrofit.callFactory(); this.callAdapter =
builder.callAdapter; this.responseConverter =
builder.responseConverter; this.baseUrl =
builder.retrofit.baseUrl(); this.relativeUrl =
builder.relativeUrl(); this.httpMethod =
builder.httpMethod; this.headers = builder.headers;
this.contentType = builder.contentType; .
this.hasBody = builder.hasBody; y this.isFormEncoded
= builder.isFormEncoded; this.isMultipart =
builder.isMultipart; this.parameterHandlers =
builder.parameterHandlers; }
```

## 步骤2: ServiceMethod的Builder ()

步骤1  
result = new ServiceMethod.Builder(this, method).build();  
步骤2  
步骤3

Paste\_Image.png

```
public Builder(Retrofit retrofit, Method method) {
 this.retrofit = retrofit; this.method = method;
 // 获取网络请求接口方法里的注释 this.methodAnnotations =
 method.getAnnotations(); // 获取网络请求接口方法里的参数
 //类型 this.parameterTypes =
 method.getGenericParameterTypes(); //获取网络请求接口
 //方法里的注解内容 this.parameterAnnotationsArray =
 method.getParameterAnnotations(); }
}
```

### 步骤3: ServiceMethod的build()

步骤1                    步骤2                    步骤3  
result = new ServiceMethod.Builder(this, method).build();

Paste\_Image.png



想  
識  
思  
學

```
// 作用：控制ServiceMethod对象的生成流程 public ServiceMethod
build() { callAdapter = createCallAdapter();
// 根据网络请求接口方法的返回值和注解类型，从Retrofit对象中获取对应的
的网络请求适配器 -->关注点1 responseType =
callAdapter.responseType(); // 根据网络请求接口方法的
返回值和注解类型，从Retrofit对象中获取该网络适配器返回的数据类型
responseConverter = createResponseConverter();
// 根据网络请求接口方法的返回值和注解类型，从Retrofit对象
中获取对应的数据转换器 -->关注点3 // 构造 HTTP 请求时，我们
传递的参数都是String // Retrofit 类提供 converter把传递的
参数都转化为 String // 其余类型的参数都利用
Converter.Factory 的stringConverter 进行转换 // @Body
和 @Part 类型的参数利用Converter.Factory 提供的
requestBodyConverter 进行转换 // 这三种 converter 都是通
过“询问”工厂列表进行提供，而工厂列表我们可以在构造 Retrofit 对象时
进行添加。 for (Annotation annotation :
methodAnnotations) {
parseMethodAnnotation(annotation); } // 解析网络
请求接口中方法的注解 // 主要是解析获取Http请求的方法 //
注解包括：DELETE、GET、POST、HEAD、PATCH、PUT、OPTIONS、HTTP、
retrofit2.http.Headers、Multipart、FormUrlEncoded //
处理主要是调用方法 parseHttpMethodAndPath(String httpMethod,
String value, boolean hasBody) ServiceMethod中的
httpMethod、hasBody、relativeUrl、relativeUrlParamNames域进
行赋值 int parameterCount =
parameterAnnotationsArray.length; // 获取当前方法的参数数
量 parameterHandlers = new ParameterHandler<?>
[parameterCount]; for (int p = 0; p <
parameterCount; p++) { Type parameterType =
parameterTypes[p]; Annotation[]
parameterAnnotations = parameterAnnotationsArray[p];
// 为方法中的每个参数创建一个ParameterHandler<?>对象并解析每
个参数使用的注解类型 // 该对象的创建过程就是对方法参数中注解
进行解析 // 这里的注解包括：Body、PartMap、Part、
FieldMap、Field、Header、QueryMap、Query、Path、Url
parameterHandlers[p] = parseParameter(p, parameterType,
parameterAnnotations); } return new
ServiceMethod<>(this);<-- 总结 -->// 1. 根据返回值类型和方法标
```

注从**Retrofit**对象的的网络请求适配器工厂集合和内容转换器工厂集合中分别获取到该方法对应的网络请求适配器和**Response**内容转换器; // 2. 根据方法的标注对**ServiceMethod**的域进行赋值// 3. 最后为每个方法的参数的标注进行解析, 获得一个**ParameterHandler<?>**对象// 该对象保存有一个**Request**内容转换器--根据参数的类型从**Retrofit**的内容转换器工厂集合中获取一个**Request**内容转换器或者一个**String**内容转换器。 }<-- 关注点1: `createCallAdapter() --> private CallAdapter<?>`

```
createCallAdapter() { // 获取网络请求接口里方法的返回值类型
 Type returnType = method.getGenericReturnType();
 // 获取网络请求接口接口里的注解 // 此处使用的是
 @Get Annotation[] annotations =
 method.getAnnotations(); try { return
 retrofit.callAdapter(returnType, annotations); // 根
 据网络请求接口方法的返回值和注解类型, 从Retrofit对象中获取对应的网
 络请求适配器 // 下面会详细说明retrofit.callAdapter () -->关注点2 }...<-- 关注点2: retrofit.callAdapter() -->
public CallAdapter<?> callAdapter(Type returnType,
Annotation[] annotations) { return
nextCallAdapter(null, returnType, annotations); } public
CallAdapter<?> nextCallAdapter(CallAdapter.Factory
skipPast, Type returnType, Annotation[] annotations)
{ // 创建 CallAdapter 如下 // 遍历
CallAdapter.Factory 集合寻找合适的工厂(该工厂集合在第一步构造
Retrofit 对象时进行添加(第一步时已经说明)) // 如果最终没有工
厂提供需要的 CallAdapter, 将抛出异常 for (int i = start,
count = adapterFactories.size(); i < count; i++) {
CallAdapter<?> adapter =
adapterFactories.get(i).get(returnType, annotations,
this); if (adapter != null) { return
adapter; } }<-- 关注点3: createResponseConverter
() --> private Converter<ResponseBody, T>
createResponseConverter() { Annotation[] annotations
= method.getAnnotations(); try { //
responseConverter 还是由 Retrofit 类提供 -->关注点4
return retrofit.responseBodyConverter(responseType,
annotations); } catch (RuntimeException e) {
throw methodError(e, "Unable to create converter for %s",
responseType); } }<-- 关注点4:
```

```
 responseBodyConverter () --> public <T>
 Converter<ResponseBody, T> responseBodyConverter(Type
 type, Annotation[] annotations) { return
 nextResponseBodyConverter(null, type, annotations); }
 public <T> Converter<ResponseBody, T>
 nextResponseBodyConverter(Converter.Factory skipPast,
 int start = converterFactories.indexOf(skipPast) + 1;
 for (int i = start, count = converterFactories.size(); i
 < count; i++) { // 获取Converter 过程: (和获取
 callAdapter 基本一致) Converter<ResponseBody, ?>
 converter =
 converterFactories.get(i).responseBodyConverter(type,
 annotations, this); // 遍历 Converter.Factory 集合并
 寻找合适的工厂 (该工厂集合在构造 Retrofit 对象时进行添加 (第一步时
 已经说明)) // 由于构造Retrofit采用的是Gson解析方式, 所
 以取出的是GsonResponseBodyConverter // Retrofit -
 Converters 还提供了 JSON, XML, ProtoBuf 等类型数据的转换功能。
 // 继续看responseBodyConverter () -->关注点5 }
<-- 关注点5: responseBodyConverter () -->@Overridepublic
 Converter<ResponseBody, ?> responseBodyConverter(Type
 type, Annotation[] annotations, Retrofit retrofit) {
 TypeAdapter<?> adapter =
 gson.getAdapter(TypeToken.get(type)); // 根据目标类型, 利用
 Gson#getAdapter 获取相应的 adapter return new
 GsonResponseBodyConverter<>(gson, adapter);}// 做数据转换时
 调用 Gson 的 API 即可。final class
 GsonResponseBodyConverter<T> implements
 Converter<ResponseBody, T> { private final Gson gson;
 private final TypeAdapter<T> adapter;
 GsonResponseBodyConverter(Gson gson, TypeAdapter<T>
 adapter) { this.gson = gson; this.adapter =
 adapter; } @Override public T convert(ResponseBody
 value) throws IOException { JsonReader jsonReader =
 gson.newJsonReader(value.charStream()); try {
 return adapter.read(jsonReader); } finally {
 value.close(); } }}
```

- 当选择了RxjavaCallAdapterFactory后，Rxjava通过策略模式选择对应的adapter

关于策略模式的讲解，请看文章[策略模式（Strategy Pattern） - 最易懂的设计模式解析](#)

- 具体过程是：根据网络接口方法的返回值类型来选择具体要用哪种CallAdapterFactory，然后创建具体的CallAdapter实例

## 采用工厂模式使得各功能模块高度解耦

- 上面提到了两种工厂：CallAdapter.Factory & Converter.Factory分别负责提供不同的功能模块
- 工厂负责如何提供、提供何种功能模块
- Retrofit 只负责提供选择何种工厂的决策信息（如网络接口方法的参数、返回值类型、注解等）

这正是所谓的高内聚低耦合，工厂模式get。

关于工厂模式请看我写的文章：

[简单工厂模式（SimpleFactoryPattern） - 最易懂的设计模式解析](#)

[工厂方法模式（Factory Method） - 最易懂的设计模式解析](#)

[抽象工厂模式（Abstract Factory） - 最易懂的设计模式解析](#)

终于配置完网络请求参数（即配置好ServiceMethod对象）。接下来将讲解第二行代码：`okHttpCall`对象的创建

\*\*第二行：`okHttpCall okHttpCall = new okHttpCall<>(serviceMethod, args);**`

根据第一步配置好的ServiceMethod对象和输入的请求参数创建`okHttpCall`对象

```
<--OkHttpCall类 -->public class OkHttpCall { private
final ServiceMethod<T> serviceMethod; // 含有所有网络请求参数
信息的对象 private final Object[] args; // 网络请求接口
的参数 private okhttp3.Call rawCall; //实际进行网络访问的
类 private Throwable creationFailure; //几个状态标志位
 private boolean executed; private volatile
boolean canceled; <--OkHttpCall构造函数 --> public
OkHttpCall(ServiceMethod<T> serviceMethod, Object[] args)
{ // 传入了配置好的ServiceMethod对象和输入的请求参数
this.serviceMethod = serviceMethod; this.args =
args; }
```

\*\*第三行： return  
serviceMethod.callAdapter.adapt(okHttpCall);\*\*

将第二步创建的okHttpCall对象传给第一步创建的serviceMethod对象  
中对应的网络请求适配器工厂的adapt()

返回对象类型：Android默认的是Call<>；若设置了  
RxJavaCallAdapterFactory，返回的则是Observable<>

```
<-- adapt () 详解-->public <R> Call<R> adapt(Call<R> call)
{ return new ExecutorCallbackCall<
(callbackExecutor, call); }
ExecutorCallbackCall(Executor callbackExecutor, Call<T>
delegate) { this.delegate = delegate; // 把上面
创建并配置好参数的okHttpCall对象交给静态代理delegate // 静
态代理和动态代理都属于代理模式 // 静态代理作用：代理执行被代理
者的方法，且可在要执行的方法前后加入自己的动作，进行对系统功能的拓展
 this.callbackExecutor = callbackExecutor;
 // 传入上面定义的回调方法执行器 // 用于进行线程切换 }
```

- 采用了**装饰模式**： ExecutorCallbackCall = 装饰者，而里面真正去执行  
网络请求的还是OkHttpCall
- 使用装饰模式的原因：希望在OkHttpCall发送请求时做一些额外操  
作。这里的额外操作是线程转换，即将子线程切换到主线程

1. OkHttpCall的enqueue()是进行网络异步请求的：当你调用 OkHttpCall.enqueue () 时，回调的callback是在子线程中，需要通过Handler转换到主线程进行回调。ExecutorCallbackCall就是用于线程回调；
2. 当然以上是原生Retrofit使用的切换线程方式。如果你用Rxjava，那就不会用到这个ExecutorCallbackCall而是RxJava的Call，此处不过多展开

**步骤4讲解：** call<JavaBean> call = NetService.getCall();

- NetService对象实际上是动态代理对象 Proxy.newProxyInstance () (步骤3中已说明)，并不是真正的网络请求接口创建的对象
- 当NetService对象调用getCall () 时会被动态代理对象 Proxy.newProxyInstance () 拦截，然后调用自身的 InvocationHandler # invoke ()
- invoke(Object proxy, Method method, Object... args)会传入3个参数：Object proxy: (代理对象)、Method method (调用的getCall()) Object... args (方法的参数，即getCall (\*) 中的\*)
- 接下来利用Java反射获取到getCall () 的注解信息，配合args参数创建serviceMethod对象。

如上面步骤3描述，此处不再次讲解

## 最终创建并返回一个OkHttpCall类型的Call对象

1. OkHttpCall类是OkHttp的包装类
2. 创建了OkHttpCall类型的Call对象还不能发送网络请求，需要创建Request对象才能发送网络请求

## 总结

Retrofit采用了外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法，具体细节是：

- 动态创建网络请求接口的实例 (**代理模式 - 动态代理**)
- 创建 serviceMethod 对象 (**建造者模式 & 单例模式 (缓存机制)**)

- 对 `serviceMethod` 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。  
**(策略模式)**
- 对 `serviceMethod` 对象加入线程切换的操作，便于接收数据后通过 Handler从子线程切换到主线程从而对返回数据结果进行处理 **(装饰模式)**
- 最终创建并返回一个`okHttpCall`类型的网络请求对象
  - \*

### 3. 执行网络请求

- Retrofit默认使用okhttp，即`okHttpCall`类（实现了`retrofit2.Call<T>`接口）

但可以自定义选择自己需要的Call类

- `okHttpCall`提供了两种网络请求方式：
  1. 同步请求：`okHttpCall.execute()`
  2. 异步请求：`okHttpCall.enqueue()`

下面将详细介绍这两种网络请求方式。

对于OkHttpCall的`enqueue()`、`execute()`此处不往下分析，有兴趣的读者可以看OkHttp的源码

#### \*\*3.1 同步请求`okHttpCall.execute()`\*\*

##### 3.1.1 发送请求过程

- **步骤1：**对网络请求接口的方法中的每个参数利用对应`ParameterHandler`进行解析，再根据`ServiceMethod`对象创建一个`okHttp`的`Request`对象
- **步骤2：**使用`okHttp`的`Request`发送网络请求；
- **步骤3：**对返回的数据使用之前设置的数据转换器(`GsonConverterFactory`)解析返回的数据，最终得到一个`Response<T>`对象

### 3.1.2 具体使用

```
Response<JavaBean> response = call.execute();
```

上面简单的一行代码，其实包含了整个发送网络同步请求的三个步骤。

### 3.1.3 源码分析

阅读理解

```
@Override public Response<T> execute() throws IOException
{ okhttp3.Call call; // 设置同步锁 synchronized (this) {
 call = rawCall; if (call == null) { try {
 call = rawCall = createRawCall(); // 步骤1: 创建
一个okhttp的Request对象请求 -->关注1 } catch
(IOException | RuntimeException e) {
 creationFailure = e; throw e; } } }
return parseResponse(call.execute()); // 步骤2: 调用
okHttpCall的execute()发送网络请求(同步) // 步骤3: 解析网络请
求返回的数据parseResponse () -->关注2}-->关注1:
createRawCall() -->private okhttp3.Call createRawCall()
throws IOException { Request request =
serviceMethod.toRequest(args); // 从ServiceMethod的
toRequest () 返回一个Request对象 okhttp3.Call call =
serviceMethod.callFactory.newCall(request); // 根据
serviceMethod和request对象创建一个okhttp3.Request if (call
== null) { throw new
NullPointerException("Call.Factory returned null."); }
return call;}-->关注2: parseResponse () -->Response<T>
parseResponse(okhttp3.Response rawResponse) throws
IOException { ResponseBody rawBody = rawResponse.body();
rawResponse = rawResponse.newBuilder() .body(new
NoContentResponseBody(rawBody.contentType(),
rawBody.contentLength())) .build(); // 收到返回数据后
进行状态码检查 // 具体关于状态码说明下面会详细介绍 int code =
rawResponse.code(); if (code < 200 || code >= 300) { }
if (code == 204 || code == 205) { return
Response.success(null, rawResponse); }
ExceptionCatchingRequestBody catchingBody = new
ExceptionCatchingRequestBody(rawBody); try { T body =
serviceMethod.toResponse(catchingBody); // 等Http请求返
后 & 通过状态码检查后, 将response body传入ServiceMethod中,
ServiceMethod通过调用Converter接口(之前设置的
ConverterFactory)将response body转成一个Java对象, 即解
析返的数据 // 生成Response类 return
Response.success(body, rawResponse); } catch
(RuntimeException e) { ... // 异常处理 } }
```

## 特别注意：

- `ServiceMethod`几乎保存了一个网络请求所需要的数据
- 发送网络请求时，`okHttpCall`需要从`ServiceMethod`中获得一个`Request`对象
- 解析数据时，还需要通过`ServiceMethod`使用`Converter`（数据转换器）转换成Java对象进行数据解析

为了提高效率，Retrofit还会对解析过的请求`ServiceMethod`进行缓存，存放在`Map<Method, ServiceMethod> serviceMethodCache = new LinkedHashMap<>();`对象中，即第二步提到的单例模式

- 关于状态码检查时的状态码说明：

| 类型               |     | 含义                                                                                                 |
|------------------|-----|----------------------------------------------------------------------------------------------------|
| 成功<br>(2开头)      | 200 | OK：请求成功、其后是对GET和POST请求的应答文档                                                                        |
|                  | 202 | Accepted：供处理的请求已被接受，但是处理未完成。                                                                       |
|                  | 204 | No Content：没有新文档。浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而Servlet可以确定用户文档足够新，这个状态代码是很有用的。                       |
|                  | 205 | Reset Content：没有新文档。但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容。                                               |
|                  | 206 | Partial Content：断点续传，客户发送了一个带有Range头的GET请求，服务器完成了它。                                                |
| 重定向<br>(3开头)     | 300 | Multiple Choices：多重选择。链接列表。用户可以选择某链接到达目的地。最多允许五个地址。                                                |
|                  | 301 | Moved Permanently：所请求的页面已经转移至新的url。                                                                |
|                  | 302 | Move temporarily：所请求的页面临时转移至新的url。如果这不是一个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认                      |
|                  | 304 | Not Modified：客户端有缓冲的文档并发出一个条件性的请求（一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。 |
| 请求错误<br>(4开头)    | 400 | Bad Request：语义有误、请求参数有误。                                                                           |
|                  | 403 | Forbidden：服务器已经理解请求，但是拒绝执行它。                                                                       |
|                  | 404 | Not Found：请求失败，请求所希望得到的资源未被在服务器上发现                                                                 |
| 服务器错误<br>(5、6开头) | 500 | Internal Server Error：请求未完成。服务器遇到不可预知的情况。                                                          |

Paste\_Image.png

以上便是整个以同步的方式发送网络请求的过程。

## 3.2 异步请求`OkHttpCall.enqueue()

### 3.2.1 发送请求过程

- **步骤1：**对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 serviceMethod 对象创建一个 okhttp 的 Request 对象
- **步骤2：**使用 okhttp 的 Request 发送网络请求；
- **步骤3：**对返回的数据使用之前设置的数据转换器（GsonConverterFactory）解析返回的数据，最终得到一个 Response<T> 对象
- **步骤4：**进行线程切换从而在主线程处理返回的数据结果

| 若使用了 RxJava，则直接回调到主线程

异步请求的过程跟同步请求类似，唯一不同之处在于：异步请求会将回调方法交回回调执行器在指定的线程中执行。

| 指定的线程此处是指主线程（UI 线程）

### 3.2.2 具体使用

```
call.enqueue(new Callback<JavaBean>() {
 @Override
 public void
 onResponse(Call<JavaBean> call, Response<JavaBean>
 response) {
 System.out.println(response.isSuccessful());
 if (response.isSuccessful()) {
 response.body().show();
 } else {
 try {
 System.out.println(response.errorBody().string());
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
 }
});
```

- 从上面分析有：call 是一个静态代理

- 使用静态代理的作用是：在okhttpCall发送网络请求的前后进行额外操作

这里的额外操作是：线程切换，即将子线程切换到主线程，从而在主线程对返回的数据结果进行处理

### 18.4.3源码分析



想  
識  
思  
學

```
<-- call.enqueue() 解析 -->@Override public void
enqueue(final Callback<T> callback) {
delegate.enqueue(new Callback<T>() { // 使用静态代理
delegate进行异步请求 ->>分析1 // 等下记得回来
@Override public void onResponse(Call<T> call,
final Response<T> response) { // 步骤4: 线程切换,
从而在主线程显示结果 callbackExecutor.execute(new
Runnable() { // 最后OkHttp的异步请求结果返回到
callbackExecutor // callbackExecutor.execute()
通过Handler异步回调将结果传回到主线程进行处理(如显示在Activity等
等), 即进行了线程切换 // 具体是如何做线程切换 ->>分析2
@Override public void run() {
if (delegate.isCanceled()) {
callback.onFailure(ExecutorCallbackCall.this, new
IOException("Canceled")); } else {
callback.onResponse(ExecutorCallbackCall.this,
response); } } });
} @Override public void onFailure(Call<T>
call, final Throwable t) {
callbackExecutor.execute(new Runnable() {
@Override public void run() {
callback.onFailure(ExecutorCallbackCall.this, t);
} }); } }<-- 分析1:
delegate.enqueue() 解析 -->@Override public void
enqueue(final Callback<T> callback) { okhttp3.Call
call; Throwable failure; // 步骤1: 创建OkHttp的Request对
象, 再封装成OkHttp.call // delegate代理在网络请求前的动作:
创建OkHttp的Request对象, 再封装成OkHttp.call synchronized
(this) { if (executed) throw new
IllegalStateException("Already executed."); executed
= true; call = rawCall; failure =
creationFailure; if (call == null && failure ==
null) { try { call = rawCall =
createRawCall(); // 创建OkHttp的Request对象, 再封
装成OkHttp.call // 方法同发送同步请求, 此处不作过多描述
} catch (Throwable t) { failure =
creationFailure = t; } } // 步骤2: 发送网络请求
// delegate是OkHttpCall的静态代理 // delegate静态代理最终还
```

```
是调用okhttp.enqueue进行网络请求 call.enqueue(new
okhttp3.Callback() { @Override public void
onResponse(okhttp3.Call call, okhttp3.Response
rawResponse) throws IOException {
Response<T> response; try { // 步
骤3: 解析返回数据 response =
parseResponse(rawResponse); } catch (Throwable e)
{ callFailure(e); return; }
callSuccess(response); } @Override
public void onFailure(okhttp3.Call call, IOException e) {
try { callback.onFailure(okHttpCall.this,
e); } catch (Throwable t) {
t.printStackTrace(); } } private void
callFailure(Throwable e) { try {
callback.onFailure(okHttpCall.this, e); } catch
(Throwable t) { t.printStackTrace(); }
} private void callSuccess(Response<T> response)
{ try {
callback.onResponse(okHttpCall.this, response); }
catch (Throwable t) { t.printStackTrace();
} }; }; } // 请回去上面分析1的起点<-- 分析2: 异步请求后
的线程切换--> // 线程切换是通过一开始创建Retrofit对象时Platform在
检测到运行环境是Android时进行创建的: (之前已分析过) // 采用适配器
模式static class Android extends Platform { // 创建默认的
回调执行器工厂 // 如果不将RxJava和Retrofit一起使用, 一般都是使
用该默认的CallAdapterFactory // 后面会对RxJava和Retrofit
一起使用的情况进行分析 @Override CallAdapterFactory
defaultCallAdapterFactory(Executor callbackExecutor) {
return new
ExecutorCallAdapterFactory(callbackExecutor); }
@Override public Executor defaultCallbackExecutor()
{ // 返回一个默认的回调方法执行器 // 该执行器负责为主线
程(UI线程)中执行回调方法 return new
MainThreadExecutor(); } // 获取主线程Handler
static class MainThreadExecutor implements Executor {
private final Handler handler = new
Handler(Looper.getMainLooper()); @Override
public void execute(Runnable r) { // Retrofit获取了
```

```

主线程的handler // 然后在UI线程执行网络请求回调后的数据显示等操作。
 handler.post(r); } } // 切换线程的流程:
程: // 1. 回调ExecutorCallAdapterFactory生成了一个
ExecutorCallbackCall对象// 2. 通过调用
ExecutorCallbackCall.enqueue(callBack)从而调用
MainThreadExecutor的execute()通过handler切换到主线程处理返回结果(如显示在Activity等等) }

```

以上便是整个以**异步方式**发送网络请求的过程。

#### 18.4.4 总结

**Retrofit** 本质上是一个 **RESTful** 的 **HTTP** 网络请求框架的封装，即通过大量的设计模式 封装了 **okhttp**，使得简洁易用。具体过程如下：

1. **Retrofit** 将 **Http** 请求 抽象 成 **Java** 接口
2. 在接口里用 **注解** 描述和配置 网络请求参数
3. 用动态代理 的方式，动态将网络请求接口的注解 解析 成 **HTTP** 请求
4. 最后执行 **HTTP** 请求

最后贴一张非常详细的 **Retrofit** 源码分析图：

| 步骤                                                                                                                                                                                                             |                                                                                                                                  | 具体过程                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                        | 使用的模式                          |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|--|--|
| 步骤1                                                                                                                                                                                                            | 创建Retrofit实例                                                                                                                     | 通过内部类Builder类建立一个Retrofit实例 ( <b>建造者模式</b> )，具体创建过程是配置了：<br>• 平台类型对象 (Platform - Android)<br>• 网络请求的url地址 (baseUrl)<br>• 网络请求工厂 (callFactory) - 默认使用OkHttpCall ( <b>工厂方法模式</b> )<br>• 网络请求适配器工厂的集合 (adapterFactories) - 默认是ExecutorCallAdapterFactory*<br>• 数据转换器工厂的集合 (converterFactories) - 本质是配置了数据转换器工厂<br>• 回调方法执行器 (callbackExecutor) - 默认回调方法执行器作用是：切换线程（子线程 - 主线程）                                                                                                                              |                                                                        | 建造者模式、工厂方法模式                   |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 步骤2                                                                                                                                                                                                            | 创建网络请求接口的实例<br>(通过解析注解配置网络请求参数)                                                                                                  | Retrofit采用了 <b>外观模式</b> 统一调用创建网络请求接口实例和网络请求参数配置的方法，具体过程：<br>1. 动态创建网络请求接口的实例 ( <b>代理模式 - 动态代理# InvocationHandler 对象# invoke ()</b> )<br>2. 创建 serviceMethod 对象 ( <b>建造者模式 &amp; 单例模式 (缓存机制)</b> )<br>3. 对 serviceMethod 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。 ( <b>策略模式</b> )<br>4. 对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过Handler从子线程切换到主线程从而对返回数据结果进行处理 ( <b>装饰模式</b> )<br>5. 最终创建并返回一个OkHttpCall类型的网络请求对象                                         |                                                                        | 外观模式、代理模式、建造者模式、单例模式、策略模式、装饰模式 |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 步骤3                                                                                                                                                                                                            | 发送网络请求                                                                                                                           | <table border="1"> <tr> <th>异步方式</th> <th>同步方式</th> </tr> <tr> <td>           1. 通过<b>静态 delegate</b>代理对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象 (<b>代理模式</b>)<br/>           2. 通过<b>静态 delegate</b>通过OkHttp.call对象发送网络请求 (<b>代理模式</b>)         </td> <td>           1. 对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象<br/>           2. 通过OkHttp.call对象发送网络请求         </td> </tr> </table> | 异步方式                                                                   | 同步方式                           | 1. 通过 <b>静态 delegate</b> 代理对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象 ( <b>代理模式</b> )<br>2. 通过 <b>静态 delegate</b> 通过OkHttp.call对象发送网络请求 ( <b>代理模式</b> ) | 1. 对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象<br>2. 通过OkHttp.call对象发送网络请求 |  |  |
| 异步方式                                                                                                                                                                                                           | 同步方式                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                        |                                |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 1. 通过 <b>静态 delegate</b> 代理对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象 ( <b>代理模式</b> )<br>2. 通过 <b>静态 delegate</b> 通过OkHttp.call对象发送网络请求 ( <b>代理模式</b> ) | 1. 对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象，并封装成OkHttp.call对象<br>2. 通过OkHttp.call对象发送网络请求 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                        |                                |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 步骤4                                                                                                                                                                                                            | 解析数据                                                                                                                             | 对返回的数据使用之前设置的数据转换器 (GsonConverterFactory) 解析返回的数据，最终得到一个 Response<T>对象                                                                                                                                                                                                                                                                                                                                                                                                                                  | 对返回的数据使用之前设置的数据转换器 (GsonConverterFactory) 解析返回的数据，最终得到一个 Response<T>对象 | 代理模式、适配器模式、装饰模式                |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 步骤5                                                                                                                                                                                                            | 切换线程                                                                                                                             | 使用回调执行器进行线程切换（子线程 - 主线程） ( <b>适配器模式、装饰模式</b> )                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <b>无线程切换</b>                                                           |                                |                                                                                                                                                                                                                |                                                                                                                                  |  |  |
| 步骤6                                                                                                                                                                                                            | 处理结果                                                                                                                             | 在主线程处理返回的数据结果                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 通过Response<T>对象处理返回的数据结果                                               |                                |                                                                                                                                                                                                                |                                                                                                                                  |  |  |

Retrofit源码分析图

# 第十九节 RxJava源码分析

理解源码之前需要先了解一些RXJava基本知识

## 19.1 RxJava操作符之创建操作符(一)

### 前言

通过前两篇文章对于RxJava概念，原理及使用的学，想必各位码友已经基本掌握RxJava的逻辑与功能了，那么从这篇文章开始我们来研究RxJava的各类操作符。

什么是操作符？通过之前的学习，我们发现Observable负责发送事件，Observer负责接收事件，而这个过程中想要对事件数据做出修改就需要交给操作符来负责啦。主流RxJava中操作符主要分为三类：转换操作符，过滤操作符，组合操作符。而我个人将用来创建Observable的操作符归为了一个新类型。本篇我们就来看看常用的创建操作符都有哪些以及如何使用。

### 创建操作符

#### Create

上一篇文章的例子中我们已经实践了create操作符如何使用了，这里我们介绍一种便捷的创建Observer的方式。

```
observable.create(new Observable.OnSubscribe<String>() {
 @Override public void call(Subscriber<? super String> subscriber) { for (int i = 0; i < 5; i++)
 { subscriber.onNext("xulei" + i); } } }.subscribe(new Action1<String>() { @Override public void call(String s) { Log.e("rx_test", "create:" + s); } });
```

subscribe()的入参使用Action1代替原来的Observer，只需重写一个call()方法，等同于原Observer中onNext()方法。如果需要onComplete与onError状态，还可以如下：

```
.subscribe(new Action1<String>() { @Override public void call(String s) { Log.e("rx_test", "create:" + s); }}, new Action1<Throwable>() { @Override public void call(Throwable throwable) { Log.e("rx_test", "onError:" + throwable.getMessage()); }}, new Action0() { @Override public void call() { Log.e("rx_test", "onCompleted"); }});
```

使用new ActionX代替new Observer，代码是不是看起来更加灵活与简洁呢。

## Just

just操作符可将某个或某些对象转化为Observable对象，并将其发射出去。参数可为一个或多个数字，字符串。也可为集合，数组，Iterate对象等。

```
observable.just(1, 2, 3, 4, 5, 6).subscribe(new Action1<Integer>() { @Override public void call(Integer integer) { Log.e("rx_test", "just:数字: " + integer); }});List<String> stringList = new ArrayList<>();stringList.add("Hello");stringList.add("Ha");stringList.add("RxJava");observable.just(stringList).subscribe(new Action1<List<String>>() { @Override public void call(List<String> strings) { Log.e("rx_test", "just:集合: " + strings.toString()); }});
```

输出结果：

```
just:数字: 1just:数字: 2just:数字: 3just:数字: 4just:数字: 5just:数字: 6just:集合: [Hello, Ha, RxJava]
```

## From

from操作符可将某个对象转化为Observable对象，并且将其发射出去。不同于just，他接收集合或数组，并可将集合数组遍历之后拆分发送。

```
List<String> stringList = new ArrayList<>()
();stringList.add("Hello");stringList.add("Ha");stringList.add("RxJava");observable.from(stringList).subscribe(new Action1<String>() {
 @Override public void call(String s) {
 Log.e("rx_test", "from: " + s);
 }
});
```

输出结果：

```
from: Hellofrom: Hafrom: RxJava
```

## Range

range(int start, int count)操作符，根据初始值start，与数量count，发射count次以start为基数依次增加的值。

```
observable.range(4, 5).subscribe(new Action1<Integer>() {
 @Override public void call(Integer integer) {
 Log.e("rx_test", "range: " + integer);
 }
});
```

输出结果：

```
range: 4range: 5range: 6range: 7range: 8
```

## Defer

defer操作符功能类似于just操作符，不同之处在于defer只有在调用subscribe()方法进行订阅时才创建Observable，而just操作符在初始化Observable就已经创建了，且只创建一个Observable实例。这里我们通过与just对比进行实践。

```
Action1<String> action1 = new Action1<String>() {
 @Override public void call(String s) {
 Log.e("rx_test", s); } };//deferObservable<String>
deferObservable = observable.defer(new
Func0<Observable<String>>() { @Override public
Observable<String> call() { Object o = new
Object(); return observable.just("defer:
hashCode: " + o.hashCode());
}});deferObservable.subscribe(action1);deferObservable.su
bscribe(action1);deferObservable.subscribe(action1);//jus
tobservable<String> justObservable =
observable.just("just: hashCode: " + new
Object().hashCode());justObservable.subscribe(action1);ju
stObservable.subscribe(action1);justObservable.subscribe(
action1);
```

输出结果：

```
defer: hashCode: 112449879defer: hashCode: 118897732defer:
hashCode: 191664429just: hashCode: 121878114just:
hashCode: 121878114just: hashCode: 121878114
```

由输出结果我们可以看出defer每次输出的Observable哈西值是不同的，说明其每subscribe订阅一次都会创建一个新的Observable，从而可保证Observable中的数据都是最新的。而just只有初始化的时候创建一次Observable。

## Interval

interval创建操作符，创建一个Observabel并每隔一段时间周期发射一个由0开始增加的数字。

**注意：此Observabel是运行在新的线程，所以更新UI需要在主线程中订阅**

```
//每隔100ms发射一个数字，从0自增observable.interval(100,
TimeUnit.MILLISECONDS) //单位为毫秒
.observeOn(AndroidSchedulers.mainThread())
.take(5) //取前5次事件发射，take为过滤操作符，后期会详细讲
.subscribe(new Action1<Long>() {
 @Override
 public void call(Long aLong) {
 Log.e("rx_test", "interval: " + aLong);
 }
});
```

输出结果：

```
100ms后...interval: 0 100ms后...interval: 1 100ms
后...interval: 2 100ms后...interval: 3 100ms后...interval: 4
```

## Timer

timer操作符，创建一个Observable并隔一段时间后发射一个特殊的值，仅发射一次。

**注意：此Observabel是运行在新的线程，所以更新UI需要在主线程中订阅**

```
//隔1s后发射一个数字observable.timer(1, TimeUnit.SECONDS) //
单位为秒 .observeOn(AndroidSchedulers.mainThread())
.subscribe(new Action1<Long>() {
 @Override public void call(Long aLong) {
 Log.e("rx_test", "timer: " + aLong);
 }
});
```

输出结果：

```
1秒后...timer: 0
```

## Delay

delay操作符，可用于延迟一定时长再发送事件。

```
//延迟2秒后发射事件observable.just(1, 2, 3) .delay(2, TimeUnit.SECONDS) .subscribe(new Action1<Integer>() { @Override public void call(Integer integer) { Log.e("rx_test", "delay: " + integer); }});
```

输出结果：

```
2秒后...delay: 1delay: 2delay: 3
```

## Repeat

repeat(long count)操作符，将Observable重复发射count次。

```
//重复发射5次“Sherlock”
observable.just("Sherlock").repeat(5)
.subscribe(new Action1<String>() {
 @Override
 public void call(String s) {
 Log.e("rx_test", "repeat: " + s);
 }
});
```

输出结果：

```
repeat: Sherlockrepeat: Sherlockrepeat: Sherlockrepeat:
Sherlockrepeat: Sherlock
```

以上就是常用的一些创建操作符，类似但不常用的还有empty、never、error等等就不一一介绍了，有兴趣的码友可以自行百度。

## 线程调度Scheduler

RxJava就是用来处理异步任务的，所以就牵扯到生产事件所在线程，处理事件所在线程的问题，下面来看一下RxJava提供的线程调度Scheduler都有哪些。

| Schedulers                     | 作用                                                                                                                                                  |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Schedulers.immediate()         | 默认的Scheduler，直接在当前线程运行                                                                                                                              |
| Schedulers.newThread()         | 总是开启一个新线程                                                                                                                                           |
| Schedulers.io()                | 用于IO密集型任务，如异步阻塞IO操作，这个调度器的线程池会根据需要增长；对于普通的计算任务，请使用Schedulers.computation(); Schedulers.io()默认是一个CachedThreadScheduler，很像一个有线程缓存的新线程调度器              |
| Schedulers.computation()       | 计算所使用的 Scheduler。这个计算指的是 CPU 密集型计算，即不会被 I/O 等操作限制性能的操作，例如图形的计算。这个 Scheduler 使用的固定的线程池，大小为 CPU 核数。不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU |
| Schedulers.from(executor)      | 使用指定的Executor作为调度器                                                                                                                                  |
| Schedulers.trampoline()        | 当其它排队的任务完成后，在当前线程排队开始执行                                                                                                                             |
| AndroidSchedulers.mainThread() | RxAndroid中新增的Scheduler，表示在Android的main线程中运行                                                                                                         |

有了Scheduler， RxJava当然也提供了方法来使用它们。

**.subscribeOn()**指定被观察者Observable的执行线程。

**.observeOn()**指定观察者Observer的执行线程。

如第一篇文章中的例子：

```
//获取要查询的小区集合
observable.from(getCommunitiesFromServer())
.flatMap(new Func1<Community, Observable<House>>() {
 @Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses()); }
 .filter(new Func1<House, Boolean>() {
 @Override public Boolean call(House
house) { return house.getPrice() < 200;
 } }) .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())
.subscribe(new Action1<House>() {
 @Override public void call(House house) {
//显示查询出来的房源信息
ShowSearchedHousesMessage();
 } });
});
```

其中 `.subscribeOn(Schedulers.io())` 指定了 Observable 在 io 线程运行，通常用来执行从服务器获取数据，数据库加载等耗时操作。`.observeOn(AndroidSchedulers.mainThread())` 指定了 Observer 在 Android 环境下的 UI 线程运行，通常用来获取到数据后进行 UI

刷新的操作。可根据实际需求选择不同线程类型。

## 总结

到此，本篇关于RxJava的创建类操作符以及线程调度就讲解完毕了，下一篇我们将一起研究RxJava的四类操作符中的转换操作符都有哪些以及如何使用。

技术渣一枚，有写的不对的地方欢迎大神们留言指正，有什么疑惑或者建议也可以在我Github上RxJavaDemo项目Issues中提出，我会及时回复。

附上RxJavaDemo的地址：

[RxJavaDemo](#)

## 19.2 RxJava操作符之转换操作符(二)

### 前言

上一篇文章我们学习了创建类操作符，本篇我们将一起来学习RxJava转换类操作符。所谓转换，就是将事件序列中的对象或整个序列进行加工处理，转换成不同的事件或事件序列。下面来看下转换类操作符都有哪些及其使用场景。

### 初始化数据

还是使用系列第一篇的小区与房源的例子。先初始化假数据以便实践操作符时使用。

```
//小区实体public class Community { private String
communityName; //小区名称 private List<House> houses; //房
源集合}//房源实体public class House { private float
size; //大小 private int floor; //楼层 private int
price; //总价 private String decoration; //装修程度
private String communityName; //小区名称}
```

```

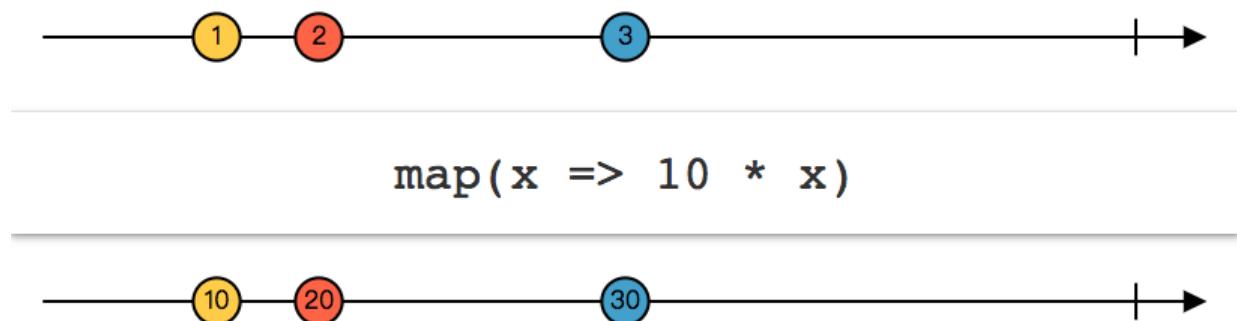
private List<Community> communities; private void
initData() { communities = new ArrayList<>();
List<House> houses1 = new ArrayList<>(); for (int i =
0; i < 5; i++) { if (i % 2 == 0) {
houses1.add(new House(105.6f, i, 200, "简单装修", "东方花
园")); } else { houses1.add(new
House(144.8f, i, 520, "豪华装修", "东方花园"));
} communities.add(new Community("东方花园", houses1));
List<House> houses2 = new ArrayList<>(); for (int
i = 0; i < 5; i++) { if (i % 2 == 0) {
houses2.add(new House(88.6f, i, 166, "中等装修", "马德里春
天")); } else { houses2.add(new
House(123.4f, i, 321, "精致装修", "马德里春天"));
} communities.add(new Community("马德里春天", houses2));
List<House> houses3 = new ArrayList<>(); for
(int i = 0; i < 5; i++) { if (i % 2 == 0) {
houses3.add(new House(188.7f, i, 724, "豪华装修", "帝
豪家园")); } else { houses3.add(new
House(56.4f, i, 101, "普通装修", "帝豪家园"));
} communities.add(new Community("帝豪家园", houses3));}

```

## 转换操作符

### Map

map操作符，接收一个指定的Func1类型对象，然后将其应用到每一个由Observable发射的值上，进而将发射的值转换为我们期望的值。来看一下原理图与实例：



```
//将一组Integer转换成String
Observable.just(1, 2, 3, 4, 5)
 .map(new Func1<Integer, String>() {
 @Override
 public String call(Integer integer) {
 return "This is " + integer;
 }
 })
 .subscribe(new Action1<String>() {
 @Override
 public void call(String s) {
 Log.e("rx_test", s);
 }
 });
//将Community集合转换为每一个Community并获取其
name
Observable.from(communities)
 .map(new Func1<Community, String>() {
 @Override
 public String call(Community community) {
 return community.getCommunityName();
 }
 })
 .subscribe(new Action1<String>() {
 @Override
 public void call(String communityName) {
 Log.e("rx_test", "小区名称
为: " + communityName);
 }
 });
});
```

输出结果：

```
This is 1
This is 2
This is 3
This is 4
This is 5
小区名称为：东方花园
小区名称为：马德里春天
小区名称为：帝豪家园
```

由输出结果可看出，`map`操作符可用来进行数据的类型转换，拼接或者对集合进行遍历等1对1的转换。第一个例子中，`Func1<Integer, String>`的第一个参数是发射数据当前的类型，第二个参数是转换之后的数据类型。`Action1<String>`中参数也为发射数据转换之后的数据类型。

**注意数据类型需对应准确，不要弄错了。**

## FlatMap

`flatMap`操作符，也是用来转换的，但与`map`操作符不同之处是，`flatMap()`返回的是`Observable`对象，且这个`Observable`对象并不是被直接发送到了`Subscriber`的回调方法中。

这么说可能不易理解，我们来看小区与房的例子，现在有3个小区，如果我们想打印出这3个小区中所有房源的信息，通过RxJava要如何做到？按照之前学习的我们或许会这么实现：

```
Observable.from(communities) .subscribe(new
Action1<Community>() { @Override
public void call(Community community) {
for (House house : community.getHouses()) {
Log.e("rx_test", "flatMap: 小区名称: " +
house.getCommunityName() + ", 价
格: " + house.getPrice() + ", 楼层: " + house.getFloor());
}
}}
```

按照这种实现方法我们只可获取到每个小区这一层，想要获取小区中的房源还需进行一层for循环遍历，这就违背了RxJava的原则了。那么来看下flatMap()如何实现：

```
Observable.from(communities) .flatMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
Observable.from(community.getHouses()); }
}) .subscribe(new Action1<House>() {
@Override public void call(House house) {
Log.e("rx_test", "flatMap: 小区名称: " +
house.getCommunityName() + ", 价格: " +
house.getPrice() + ", 楼层: " + house.getFloor());
}
});
```

这样的代码是不是看起来舒心多了，再来看下flatMap()是如何实现的。

首先from()接收到小区集合communities后为其创建了一个Observable，依次将每个小区传递给flatMap()，flatMap()在每次接收到小区后会将其中包含的房源集合拿出来又创建了一个房源Observable，并激活这个房源Observable让其开始发射事件，之后返回给小区集合的Observable，最后小区集合的Observable再将这些事件统一交给Subscriber的回调方法去处理。

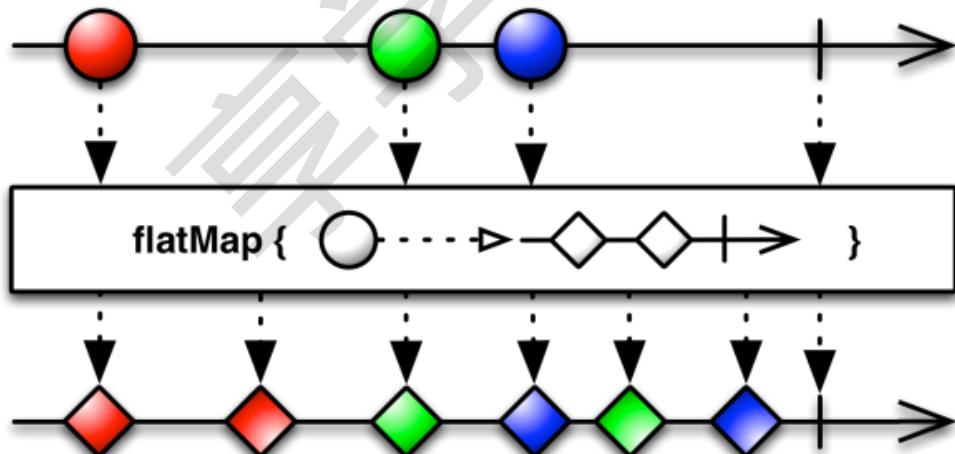
整个过程有两级Observable在运作，相当于将小区集合Observable这个初始对象铺平之后再通过统一路径分发下去，**铺平**这个工作就是flatMap所做的。

输出结果：

```
flatMap: 小区名称: 东方花园, 价格: 200, 楼层: 0
flatMap: 小区名称: 东方花园, 价格: 520, 楼层: 1
flatMap: 小区名称: 东方花园, 价格: 200, 楼层: 2
flatMap: 小区名称: 东方花园, 价格: 520, 楼层: 3
flatMap: 小区名称: 东方花园, 价格: 200, 楼层: 4
flatMap: 小区名称: 马德里春天, 价格: 166, 楼层: 0
flatMap: 小区名称: 马德里春天, 价格: 321, 楼层: 1
flatMap: 小区名称: 马德里春天, 价格: 166, 楼层: 2
flatMap: 小区名称: 马德里春天, 价格: 321, 楼层: 3
flatMap: 小区名称: 帝豪家园, 价格: 724, 楼层: 0
flatMap: 小区名称: 帝豪家园, 价格: 101, 楼层: 1
flatMap: 小区名称: 帝豪家园, 价格: 724, 楼层: 2
flatMap: 小区名称: 帝豪家园, 价格: 101, 楼层: 3
flatMap: 小区名称: 帝豪家园, 价格: 724, 楼层: 4
```

由输出结果可看出这3个小区的所有房源信息都被依次打印了出来，但 flatMap()有一个问题就是当数据量过大时可能会出现输出数据顺序交错的问题。

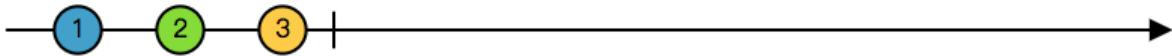
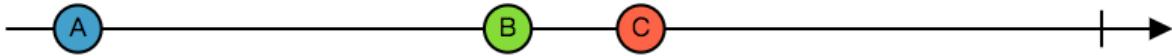
官方原理图：



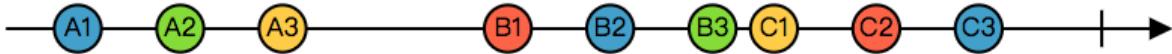
## ConcatMap

concatMap操作符，与flatMap()功能类似。不同之处是concatMap()采用连接方式而不是合并方式，所以其发射的数据是严格按照顺序的，这就解决了flatMap()有可能发生数据交错的问题。

原理图：



```
obs1$.concatMap(() => obs2$, (x, y) => "" + x + y)
```



## FlatMapIterable

flatMapIterable操作符，也与flatMap()相似，不同之处在于flatMapIterable转化多个Observable是使用Iterable作为源数据的。

```
Observable.from(communities).flatMapIterable(new Func1<Community, Iterable<House>>() {
 @Override
 public Iterable<House> call(Community community) {
 return community.getHouses();
 }
}).subscribe(new Action1<House>() {
 @Override
 public void call(House house) {
 Log.e("rx_test",
 "flatMap: 小区名称: " + house.getCommunityName()
 + ", 价格: " + house.getPrice() + ", 楼层: " +
 house.getFloor());
 }
});
```

## SwitchMap

switchMap转换操作符，也与flatMap()相似，每当源Observable发射新数据项(Observable)时，它将取消订阅并停止监视之前那个数据项产生的Observable，并开始监视当前发射的这一个。

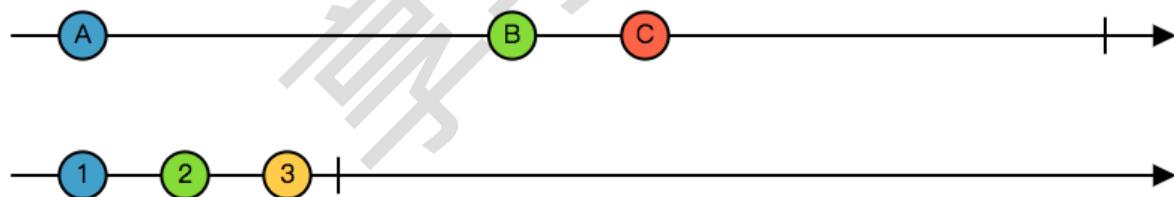
```

Observable.from(communities) .switchMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses()); }
}) .subscribe(new Action1<House>() {
@Override public void call(House house) {
Log.e("rx_test", "flatMap: 小区名称: " +
house.getCommunityName() + ", 价格: " +
house.getPrice() + ", 楼层: " + house.getFloor());
}
});

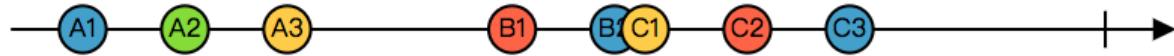
```

如之前的例子，当数据量很大时，某一时刻，第一个小区所生成的小房源 Observable正在发射数据，这时第二个小区所生成的小房源 Observable 被激活，则第一个小区的小 Observable 就会被取消订阅，其还未发射的数据也不在发射了。第二个小区小 Observable 开始发射数据，之后都同理。

原理图：



```
obs1$.switchMap(() -> obs2$, (x, y) -> "" + x + y)
```



## Scan

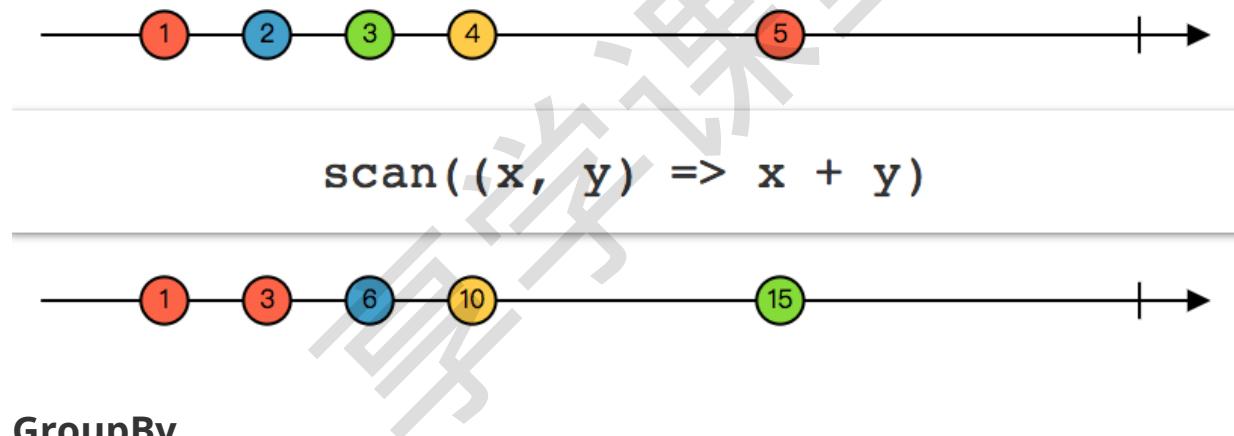
scan操作符，对一个序列的数据应用一个函数，并将这个函数的结果发射出去作为下个数据应用函数时的第一个参数使用。

```
//例如：先输出1，再将1+2=3作为下个数据发出，3+3=6再作为下个数据发出，以此类推。Observable.just(1, 2, 3, 4, 5)
 .scan(new Func2<Integer, Integer, Integer>() {
 @Override
 public Integer call(Integer integer,
 Integer integer2) {
 return integer +
 integer2;
 }
 })
 .subscribe(new Action1<Integer>() {
 @Override
 public void call(Integer integer) {
 Log.e("rx_test", "scan: " + integer);
 }
 });
}
```

输出结果：

```
scan: 1
scan: 3
scan: 6
scan: 10
scan: 15
```

原理图：



## GroupBy

groupBy操作符，将原始Observable发射的数据按照key来拆分成一些小的Observable，然后这些小的Observable分别发射其所包含的数据。通俗的说就是按照某个字段将数据进行分类再发射。

来看一个例子：有几个小区的多套房源数据，现在需要将其按照小区名称进行分类并输出。

```

List<House> houseList = new ArrayList<>
() ;houseList.add(new House(105.6f, 1, 200, "简单装修", "东方花园"));houseList.add(new House(144.8f, 3, 300, "豪华装修", "马德里春天"));houseList.add(new House(88.6f, 2, 170, "简单装修", "东方花园"));houseList.add(new House(123.4f, 1, 250, "简单装修", "帝豪家园"));houseList.add(new
House(144.8f, 6, 350, "豪华装修", "马德里春
天"));houseList.add(new House(105.6f, 4, 210, "普通装修",
"东方花园"));houseList.add(new House(188.7f, 3, 400, "精致装
修", "帝豪家园"));houseList.add(new House(88.6f, 2, 180,
"普通装修", "东方花园"));//根据小区名称进行分类
Observable<GroupedObservable<String, House>>
groupByCommunityNameObservable = Observable
.from(houseList) .groupBy(new Func1<House, String>
() { @Override public String
call(House house) { //提供分类规则的key
return house.getCommunityName(); }
});Observable.concat(groupByCommunityNameObservable)
//concat组合操作符，将多个Observable有序组合并发送，后期会详细讲
解 .subscribe(new Action1<House>() {
@Override public void call(House house) {
Log.e("rx_test", "groupBy: " + "小区: " +
house.getCommunityName() + ", 价格: " + house.getPrice());
}
});

```

创建一个新的Observable: groupByCommunityNameObservable，它将会发送一个带有GroupedObservable的序列（也就是指发送的数据项的类型为GroupedObservable）。GroupedObservable是一个特殊的Observable，它基于一个分组的key，在这个例子中的key就是小区名。

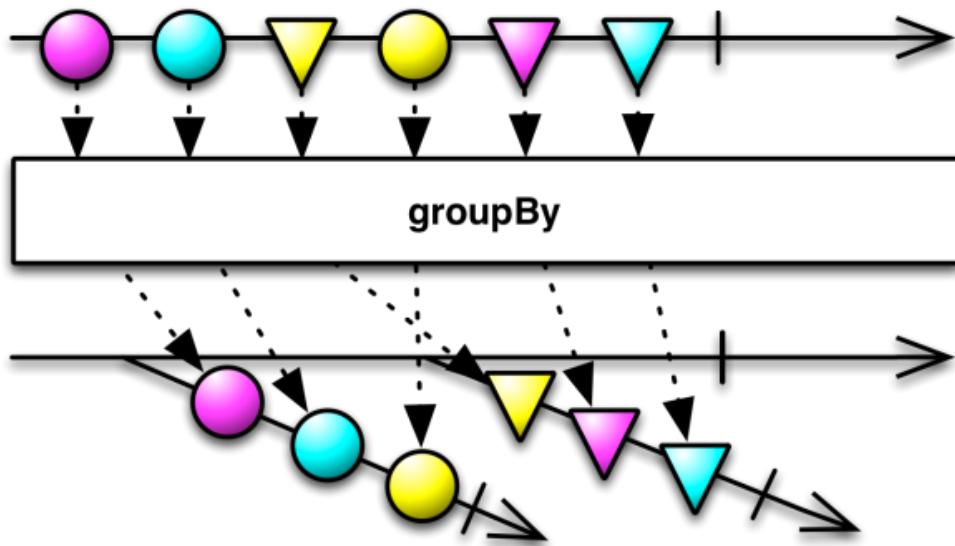
输出结果：

```

groupBy: 小区: 东方花园, 价格: 200
groupBy: 小区: 东方花园, 价格:
170
groupBy: 小区: 东方花园, 价格: 210
groupBy: 小区: 东方花园, 价
格: 180
groupBy: 小区: 马德里春天, 价格: 300
groupBy: 小区: 马德里
春天, 价格: 350
groupBy: 小区: 帝豪家园, 价格: 250
groupBy: 小区:
帝豪家园, 价格: 400

```

原理图：



## 总结

到此，本篇关于RxJava的常用转换类操作符就讲解完毕了，下一篇我们将一起研究RxJava的四类操作符中的过滤操作符都有哪些以及如何使用。

技术渣一枚，有写的不对的地方欢迎大神们留言指正，有什么疑惑或者建议也可以在我Github上RxJavaDemo项目Issues中提出，我会及时回复。

附上RxJavaDemo的地址：

[RxJavaDemo](#)

## 19.3 RxJava操作符之过滤操作符(三)

### 前言

上一篇文章我们学习了转换类操作符，本篇我们将一起来学习RxJava过滤类操作符。过滤操作符主要是用来对事件数据进行过滤与筛选，只返回满足条件的数据，一起来看下都有哪些。

### 过滤操作符

`filter`操作符，按照一定的约束条件过滤序列中我们不想要的数据，只返回满足条件的数据给观察者。

```

//结合flatMap，过滤出各小区中房源大小大于120平的房子
observable.from(communities) .flatMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses()); }
}) .filter(new Func1<House, Boolean>() {
@Override public Boolean call(House
house) { return house.getSize() > 120f;
}
}) .subscribe(new Action1<House>
() {
@Override public void
call(House house) { Log.e("rx_test",
"filter: 大于120平的房子: " + house.getCommunityName() + "小
区, 大小: " + house.getSize()); }
});

```

由代码可见，我们需要new一个Func1对象给filter(), Func1<House, Boolean>() 中第一个是由观测序列传入数据的类型，第二个是返回是否过滤的Boolean对象。满足filter()的条件则返回true，否则返回false。并将返回为true的数据发射给观察者。

输出结果：

```

filter: 大于120平的房子: 东方花园小区, 大小: 144.8filter: 大于
120平的房子: 东方花园小区, 大小: 144.8filter: 大于120平的房子: 马
德里春天小区, 大小: 123.4filter: 大于120平的房子: 马德里春天小区,
大小: 123.4filter: 大于120平的房子: 帝豪家园小区, 大小:
188.7filter: 大于120平的房子: 帝豪家园小区, 大小: 188.7filter:
大于120平的房子: 帝豪家园小区, 大小: 188.7

```

原理图：



`filter(x => x > 10)`



实际项目开发中，filter操作符可用来过滤数据集合中的null值，方便实用。

## Take

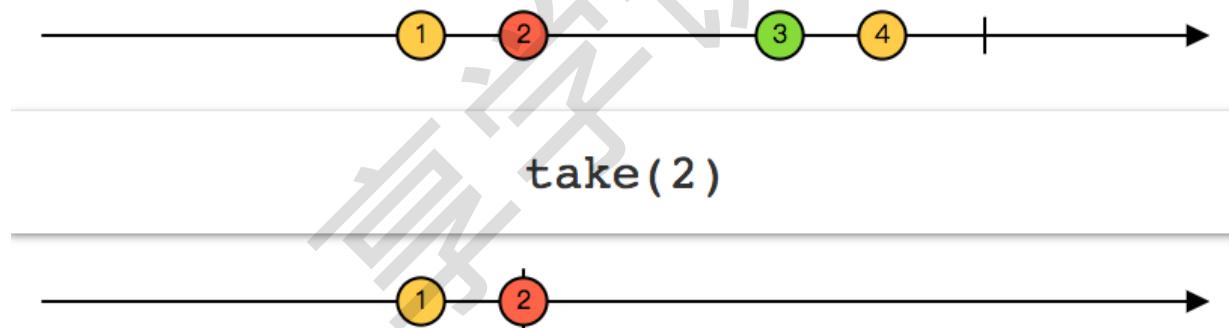
take(int count)操作符，可用来截取观测序列中前count个元素并发射。

```
//take: 获取前两个小区名
observable.from(communities)
 .take(2)
 .subscribe(new Action1<Community>() {
 @Override
 public void call(Community community) {
 Log.e("rx_test", "take: 前两个小区: " + community.getCommunityName());
 }
 });
 }
```

输出结果：

```
take: 前两个小区: 东方花园
take: 前两个小区: 马德里春天
```

原理图：



## TakeLast

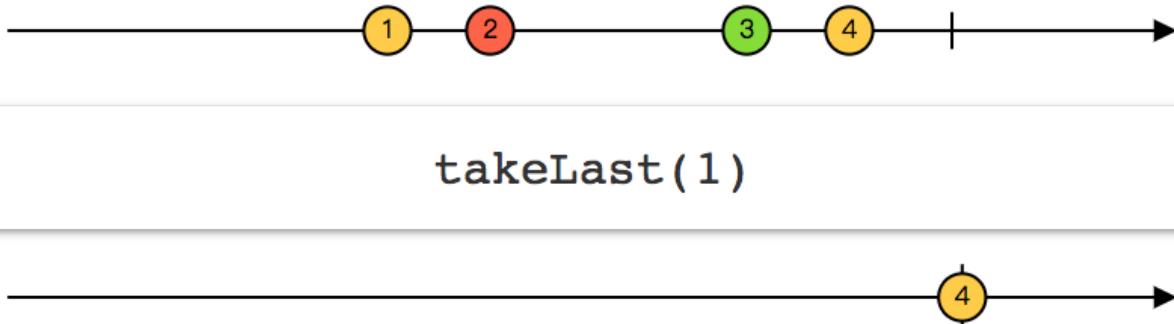
takeLast(int count)操作符，顾名思义，截取观测序列中后count个元素并发射。

```
//takeLast: 获取后两个小区名
observable.from(communities)
 .takeLast(2)
 .subscribe(new Action1<Community>
() {
 @Override
 public void
call(Community community) {
 Log.e("rx_test", "takeLast: 后两个小区: " +
community.getCommunityName());
}
});
```

输出结果：

takeLast: 后两个小区：马德里春天 takeLast: 后两个小区：帝豪家园

原理图：



takeUntil操作符有两种类型的入参。

### 1. `takeUntil(Observable)`

订阅并开始发射原始Observable，同时监视我们提供的第二个Observable。如果第二个Observable发射了一项数据或者发射了一个终止通知，`takeUntil()`返回的Observable会停止发射原始Observable并终止。

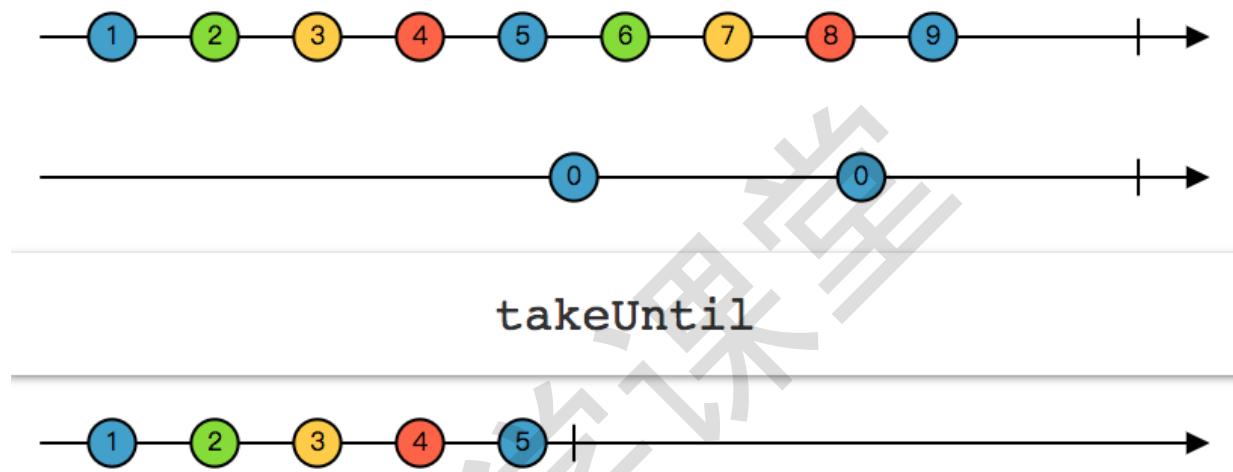
```
//observableA每300ms发射一个Long型自增数据//observableB每
800ms发射一个Long型自增数据Observable<Long> observableA =
Observable.interval(300,
TimeUnit.MILLISECONDS);Observable<Long> observableB =
Observable.interval(800,
TimeUnit.MILLISECONDS);observableA.takeUntil(observableB)
.subscribe(new Subscriber<Long>() {
@Override
public void onCompleted() {
Log.e("rx_test", "takeUntil(Observable): " +
"onCompleted");
}
@Override
public void onError(Throwable e) {
Log.e("rx_test", "takeUntil(Observable): " +
"onError: " + e.getMessage());
}
@Override
public void onNext(Long aLong) {
Log.e("rx_test",
"takeUntil(Observable): onNext: " + aLong);
}
});
```

输出结果：

```
takeUntil(observable): onNext: 0takeUntil(observable):
onNext: 1takeUntil(observable): onCompleted
```

由输出结果可看出，订阅之后，observableA依次发射0, 1之后就发射onCompleted标记停止了。这是由于observableA每300ms发射一次，当发射完1后，时间已过去600ms，到800ms时observableB开始发射数据，takeUntil起作用则中断了observableA的发射。

原理图：



## 2. `takeUntil(Func1)`

通过传入的Func1中的call()方法判断是否中止发射数据。

```

//takeUntil: 与flatMap结合过滤直到房价大于500时中断当前小
observable发射HouseObservable.from(communities)
.flatMap(new Func1<Community, Observable<House>>() {
 @Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses()); }
}) .takeUntil(new Func1<House, Boolean>() {
 @Override public Boolean call(House house) { return house.getPrice() > 500; }
}) .subscribe(new Action1<House>
() {
 @Override public void call(House house) {
 Log.e("rx_test",
"takeUntil: 大于500时中断发射: " + house.getCommunityName()
+ "小区, 房价: " + house.getPrice()); }
});

```

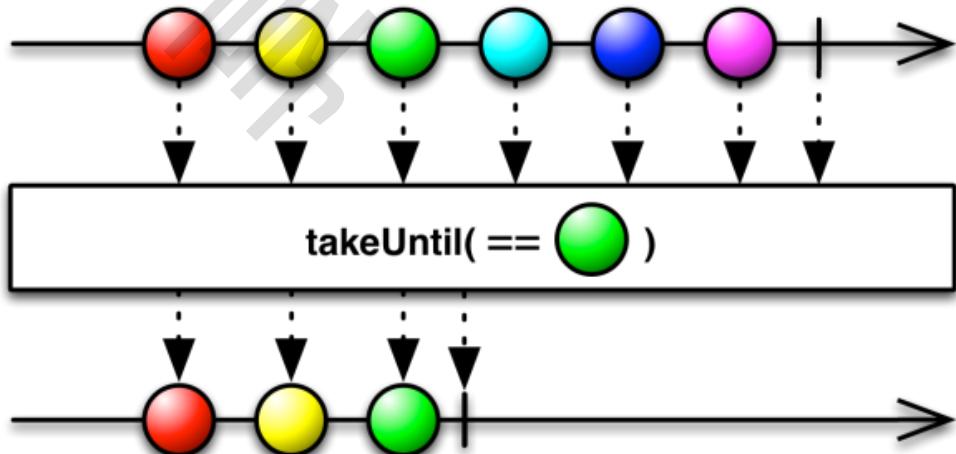
输出结果：

```

takeUntil(Func1): 大于500时中断发射: 东方花园小区, 房价:
200takeUntil(Func1): 大于500时中断发射: 东方花园小区, 房价: 520

```

原理图：



## TakeWhile

`takeWhile`操作符，类似于`takeUntil(Func1)`，不过`takeWhile()`是当Observable发射的数据不满足条件时中止Observable的发射。

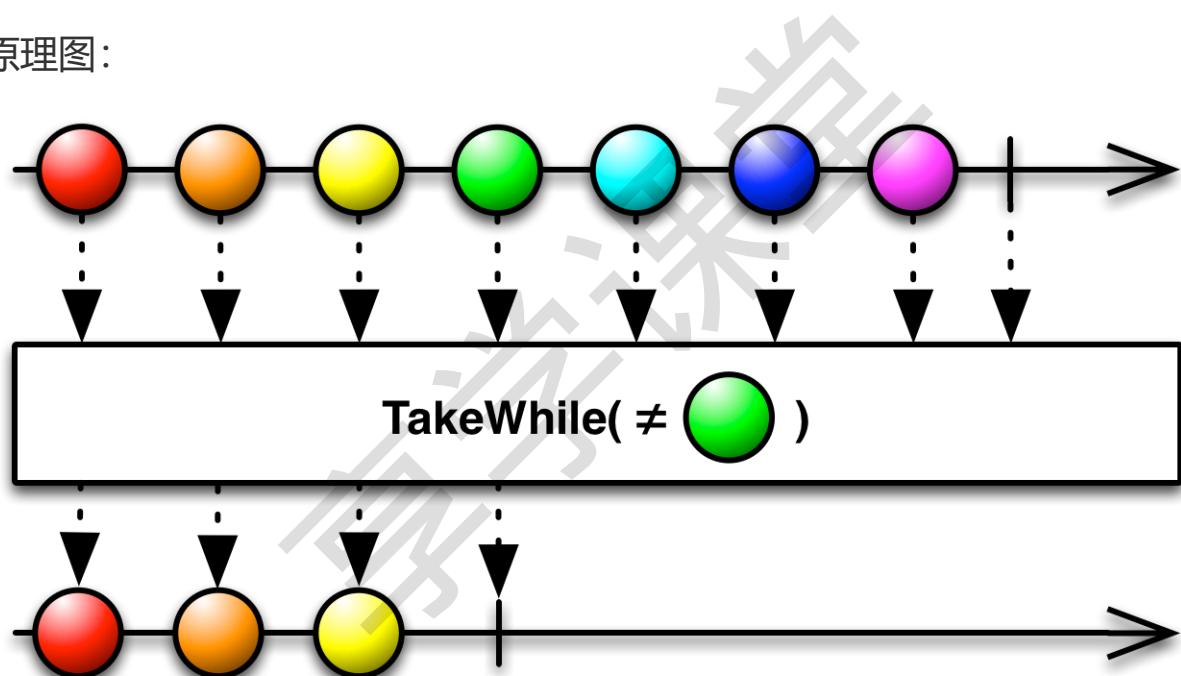
```

//takewhile: 当发射的数据等于3时中止发射Observable.just(1, 2,
3, 4, 5) .takewhile(new Func1<Integer, Boolean>()
{
 @Override public Boolean
call(Integer integer) { return integer !=
3; } } .subscribe(new
Action1<Integer>() { @Override
public void call(Integer integer) {
Log.e("rx_test", "takewhile: " + integer); }
});
```

输出结果：

```
takewhile: 1takewhile: 2
```

原理图：



## Skip

skip(int count)操作符，忽略发射观测序列的前count项数据。

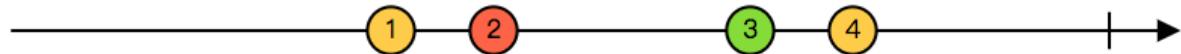
```

//忽略前两个小区数据Observable.from(communities)
.skip(2) .subscribe(new Action1<Community>() {
 @Override public void call(Community
community) { Log.e("rx_test", "skip: 忽略前
两个小区: " + community.getCommunityName()); }
});
```

输出结果：

**skip:** 忽略前两个小区：帝豪家园

原理图：



**skip(2)**



## SkipLast

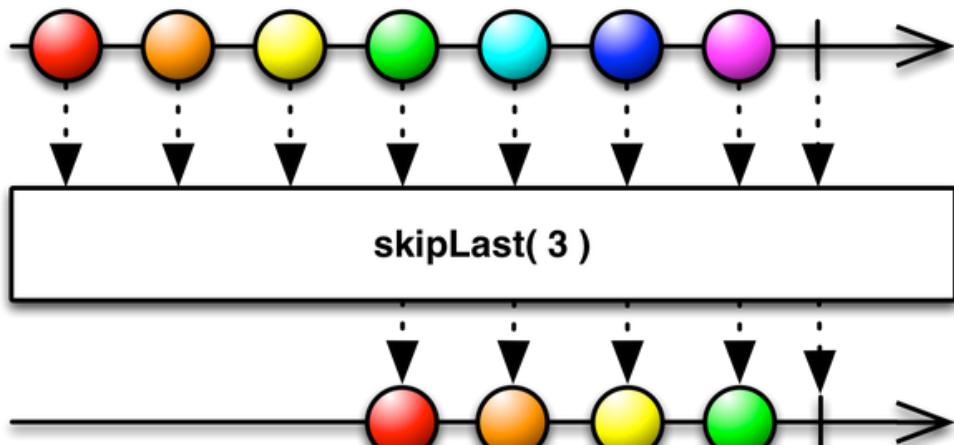
skipLast(int count)操作符，忽略发射观测序列的后count项数据。

```
//忽略后两个小区数据
Observable.from(communities)
 .skipLast(2) .subscribe(new Action1<Community>() {
 @Override public void
 call(Community community) {
 Log.e("rx_test", "skip: 忽略后两个小区: " +
 community.getCommunityName()); }
 });
}
```

输出结果：

忽略后两个小区：东方花园

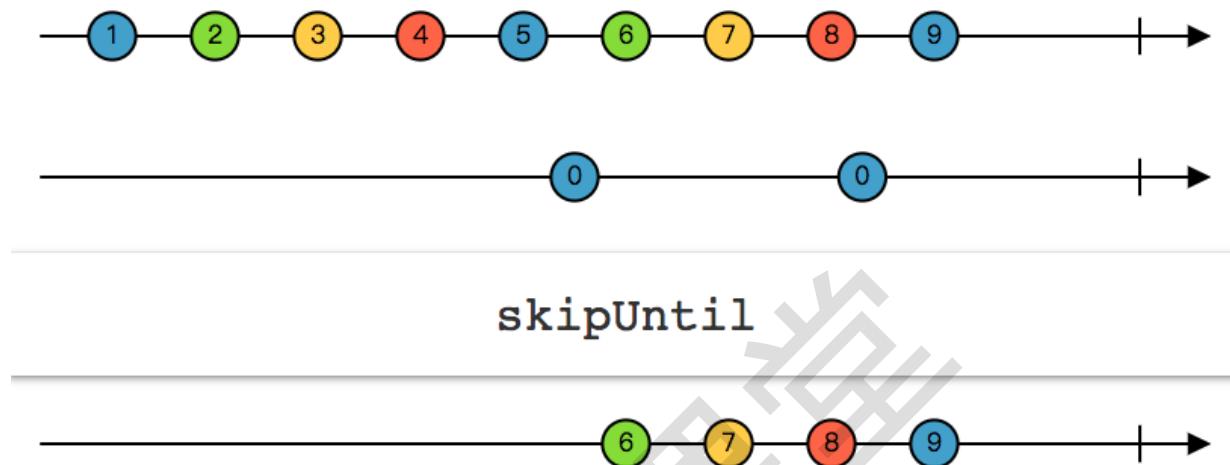
原理图：



## SkipUntil

skipUntil操作符，与takeUntil()相反。订阅并开始发射原始Observable，同时监视我们提供的第二个Observable。如果第二个Observable发射了一项数据或者发射了一个终止通知，skipUntil()返回的Observable才会开始发射数据，忽略之前的数据项。

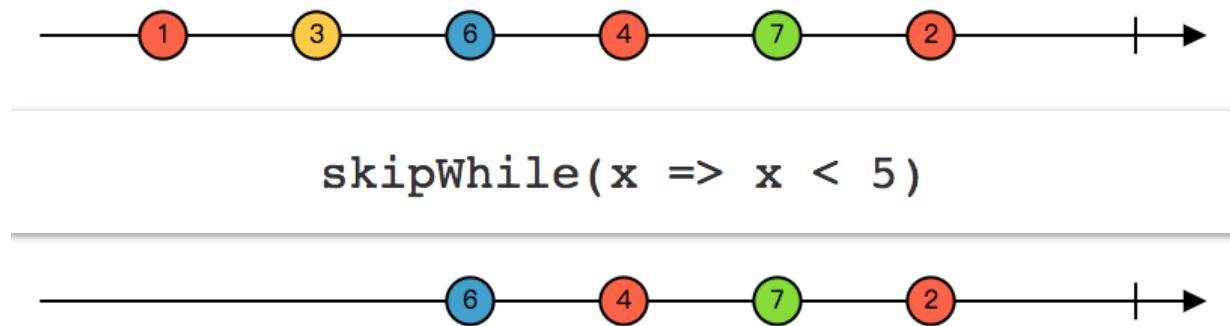
原理图：



## SkipWhile

skipWhile操作符，与takeWhile相反，当Observable发射的数据不满足条件时才开始发射数据，忽略之前的数据项。

原理图：



## Debounce

debounce操作符有两种类型的入参。

## 1.debounce(long, TimeUnit)

过滤由Observable发射的速率过快的数据，起到限流的作用。第一个参数为限流时间，第二个参数为时间单位。

```
observable.create(new Observable.OnSubscribe<Integer>() {
 @Override public void call(Subscriber<? super Integer> subscriber) { try { for (int i = 1; i < 10; i++) { subscriber.onNext(i); Thread.sleep(i * 100); //分别延时100, 200, 300, 400, 500.....900ms发射数据 } } catch (InterruptedException e) { e.printStackTrace(); } } }).subscribeOn(Schedulers.newThread()) .debounce(400, TimeUnit.MILLISECONDS) .subscribe(new Observer<Integer>() {
 @Override public void onCompleted() { Log.e("rx_test", "debounce: " + "onCompleted"); }
 @Override public void onError(Throwable e) { }
 @Override public void onNext(Integer integer) { Log.e("rx_test", "debounce: " + integer); }
});
```

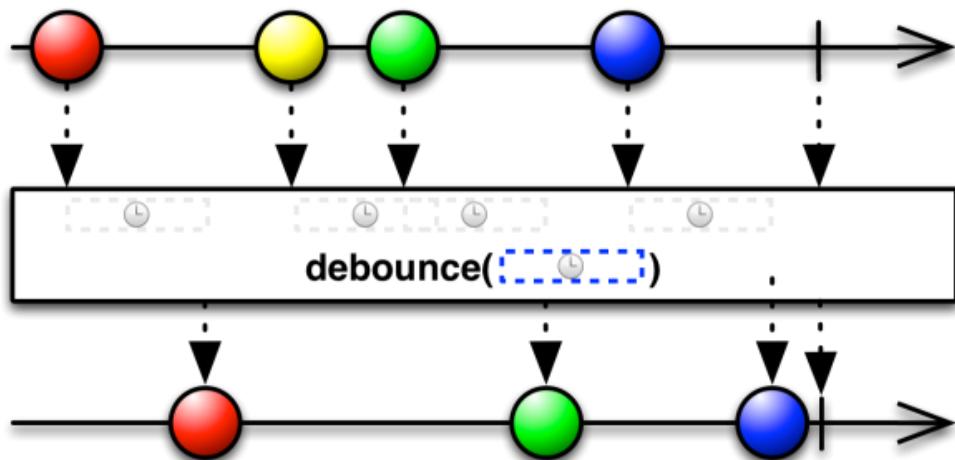
输出结果：

```
debounce: 5debounce: 6debounce: 7debounce: 8debounce:
9debounce: onCompleted
```

由输出结果可以看出由于设定限流时间为500ms，所以1-4并没有被发射而是被过滤了。

**注意：如果源Observable产生的最后一个结果在限流时间内内调用了onCompleted，那么通过debounce操作符也会把这个结果提交给订阅者。**

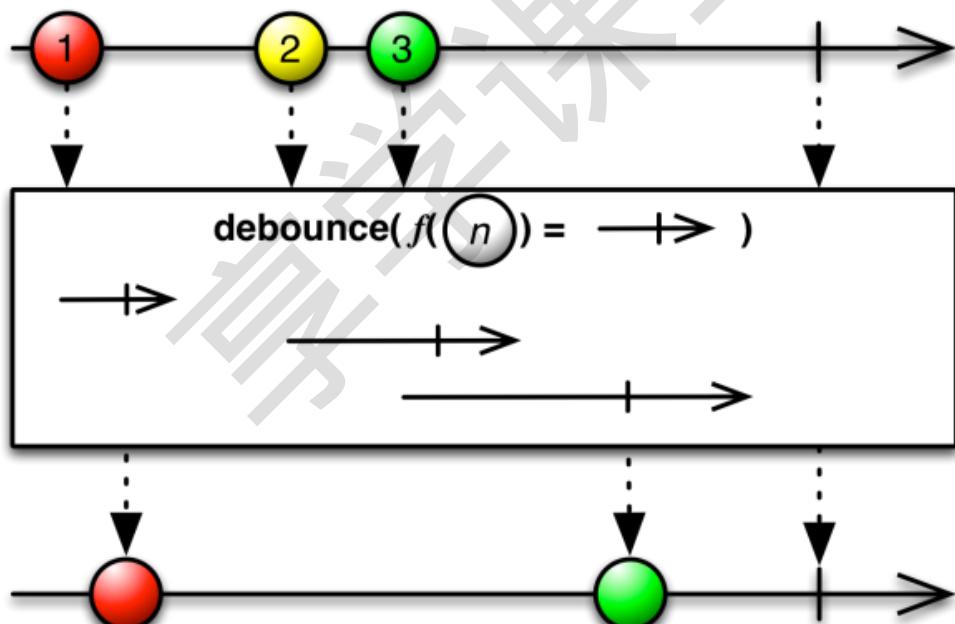
原理图：



## 2. debounce(Func1)

根据Func1的call方法中的函数来过滤。Func1中的call方法返回了一个临时的Observable，如果原始的Observable在发射一个新的数据时，上一个数据根据Func1的call方法生成的临时Observable还没结束，那么上一个数据就会被过滤掉。

原理图：



## Distinct

### 1.distinct()

只允许还没有发射过的数据通过，达到去除序列中重复项的作用。

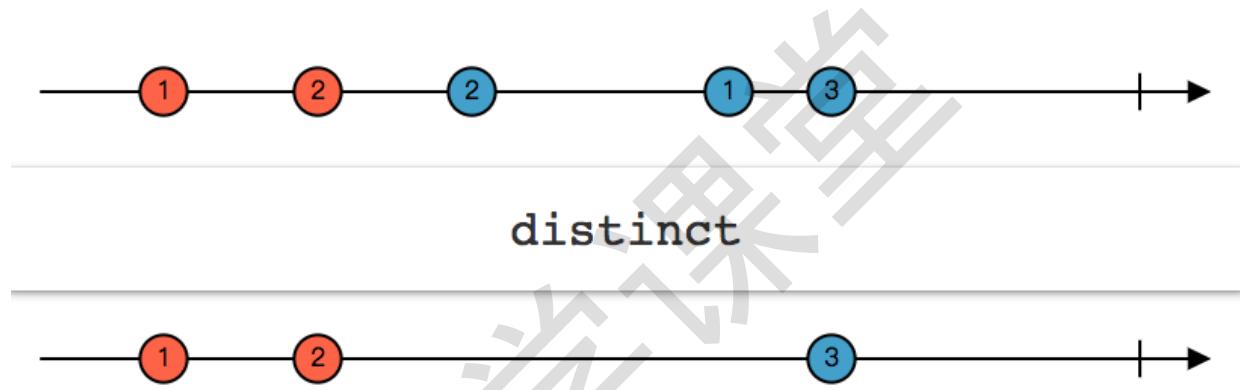
```
//去除重复数字
Observable.just(1, 2, 2, 3, 4, 5, 6, 6, 6, 7)
 .distinct() .subscribe(new Action1<Integer>
() { @Override public void
call(Integer integer) { Log.e("rx_test",
"distinct: 去重: " + integer); }
});
```

输出结果：

```
distinct: 去重: 1distinct: 去重: 2distinct: 去重: 3distinct:
去重: 4distinct: 去重: 5distinct: 去重: 6distinct: 去重: 7
```

由输出结果可见有重复的2和6都被过滤了。

原理图：



## 2. `distinct(Func1)`

根据`Func1`中的`call`方法进行去重，`call`方法会根据`Observable`发射的值生成一个Key，然后比较这个key来判断两个数据是否相同，如果判定为重复则会和`distinct()`一样过滤掉重复的数据项。

```

//根据某属性去重，去除各小区大小相同的房源
Observable.from(communities) .flatMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
Observable.from(community.getHouses()); }
}) .distinct(new Func1<House, Float>() {
@Override public float call(House house)
{
return house.getSize(); }
}) .subscribe(new Action1<House>() {
@Override public void call(House house) {
Log.e("rx_test", "distinct(Func1): 去重: " +
house.getCommunityName() + "小区，大小: " +
house.getSize()); }
});
```

输出结果：

```

distinct(Func1): 去重: 东方花园小区，大小:
105.6distinct(Func1): 去重: 东方花园小区，大小:
144.8distinct(Func1): 去重: 马德里春天小区，大小:
88.6distinct(Func1): 去重: 马德里春天小区，大小:
123.4distinct(Func1): 去重: 帝豪家园小区，大小:
188.7distinct(Func1): 去重: 帝豪家园小区，大小: 56.4
```

## DistinctUntilChanged

### 1.distinctUntilChanged()

通过当前数据项与前一项是否相同来进行去重。

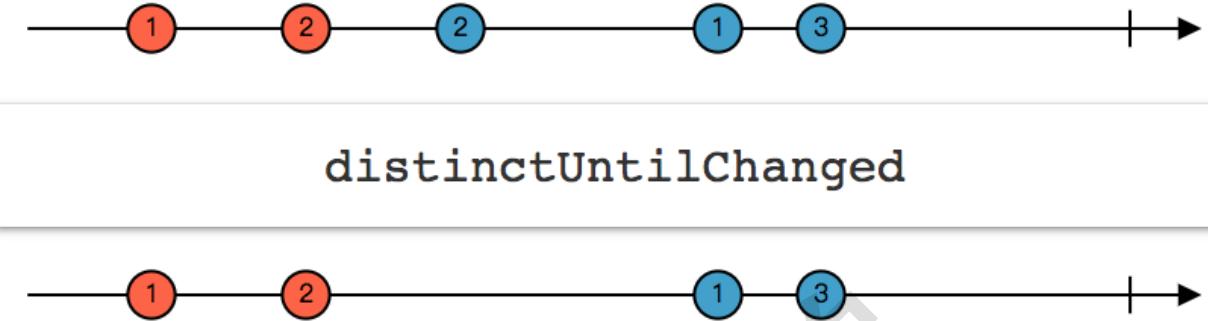
```

//向前去重复数据Observable.just(1, 2, 2, 3, 4, 2, 3, 5, 5)
 .distinctUntilChanged() .subscribe(new
Action1<Integer>() { @Override
public void call(Integer integer) {
Log.e("rx_test", "distinctUntilChanged: 向前去重: " +
integer); }
});
```

输出结果：

```
distinctUntilChanged: 向前去重: 1
distinctUntilChanged: 向前去重: 2
distinctUntilChanged: 向前去重: 3
distinctUntilChanged: 向前去重: 4
distinctUntilChanged: 向前去重:
2
distinctUntilChanged: 向前去重: 3
distinctUntilChanged: 向前去重: 5
```

原理图:



## 2.distinctUntilChanged(Func1)

与distinct(Func1)类似，根据Func1中call方法产生一个key来判断相邻两个数据项是否相同。

```
//根据某属性向前去重，去除各小区名相同的房源
observable.from(communities) .flatMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses())
.distinctUntilChanged(new Func1<House, String>() {
 @Override public String
call(House house) { return
house.getCommunityName();
}); } }) .subscribe(new
Action1<House>() { @Override public
void call(House house) { Log.e("rx_test",
"distinctUntilChanged(Func1): 向前去重: " +
house.getCommunityName() + "小区，大小: " +
house.getSize()); } });
});
```

输出结果:

```
distinctUntilChanged(Func1): 向前去重: 东方花园小区, 大小:
105.6distinctUntilChanged(Func1): 向前去重: 马德里春天小区, 大
小: 88.6distinctUntilChanged(Func1): 向前去重: 帝豪家园小区, 大
小: 188.7
```

## ElementAt

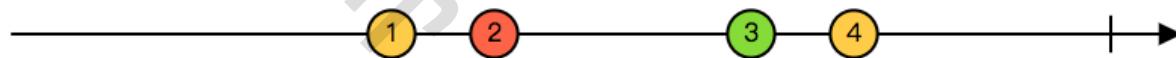
elementAt(int index)操作符，获取观测序列中第index项索引，并作为唯一数据发射给观察者，index索引从0开始。

```
observable.from(communities).elementAt(1)
.subscribe(new Action1<Community>() {
@Override
public void call(Community
community) {
Log.e("rx_test", "elementAt:
第二个小区: " + community.getCommunityName());
}});
```

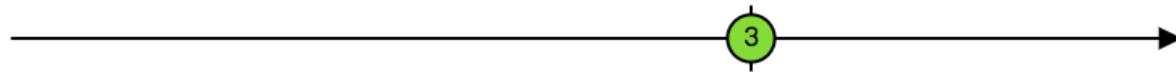
输出结果：

```
elementAt: 第二个小区: 马德里春天
```

原理图：



elementAt(2)



## First

### 1.first()

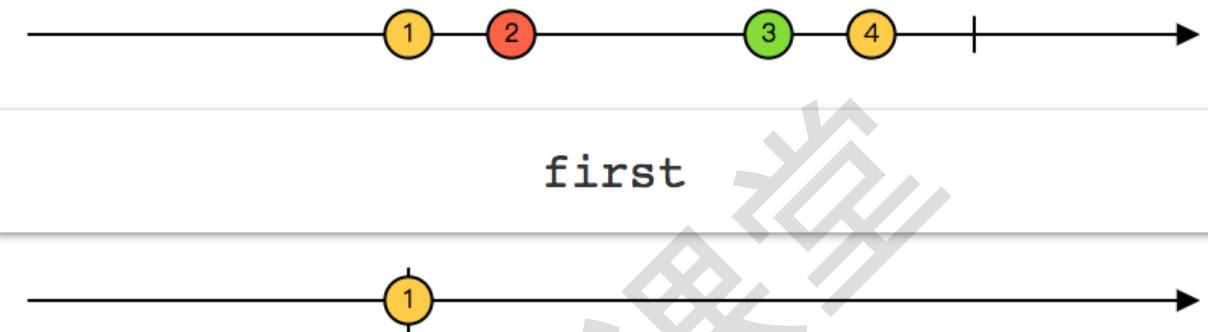
只发射观测序列中的第一个数据项。

```
Observable.from(communities) .first()
.subscribe(new Action1<Community>() {
 @Override
 public void call(Community
community) {
 Log.e("rx_test", "first: " +
community.getCommunityName());
 }
});
```

输出结果：

```
first: 东方花园
```

原理图：



## 2. `first(Func1)`

根据Func1中call方法的条件，发射符合条件的第一个数据项。

```
//过滤出第一个小区名为马德里春天的小区
Observable.from(communities) .first(new
Func1<Community, Boolean>() {
 @Override
 public Boolean call(Community community) {
 return "马德里春
天".equals(community.getCommunityName());
 }
}).subscribe(new Action1<Community>() {
 @Override
 public void call(Community
community) {
 Log.e("rx_test",
"first(Func1): " + community.getCommunityName());
 }
});
```

输出结果：

```
first(Func1): 马德里春天
```

## Last

### 1.last()

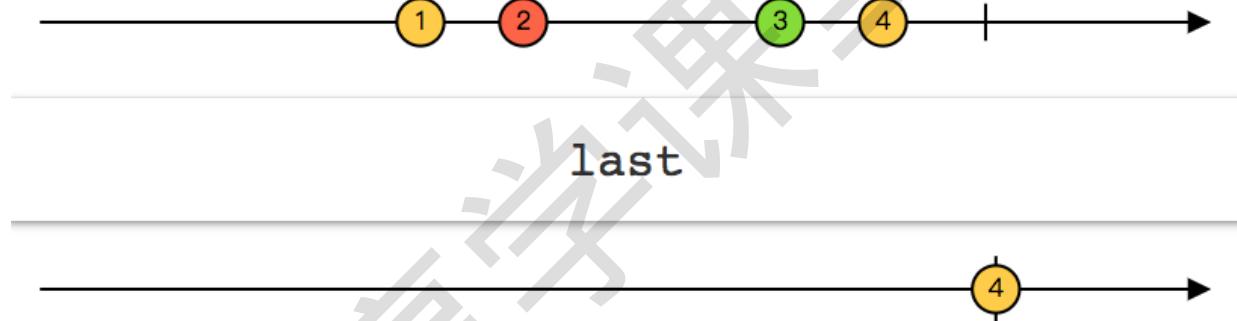
只发射观测序列中的最后一个数据项。

```
//发送最后一个数据项
Observable.from(communities)
 .last()
 .subscribe(new Action1<Community>() {
 @Override
 public void call(Community
 community) {
 Log.e("rx_test", "last: " +
 community.getCommunityName());
 }
 });
 }
```

输出结果：

```
last: 帝豪家园
```

原理图：



### 2.last(Func1)

根据Func1中call方法的条件，发射符合条件的最后一个数据项。

```
//发送符合条件的最后一个数据项：过滤最后一个小区名为马德里春天的房源
observable.from(communities) .flatMap(new
Func1<Community, Observable<House>>() {
@Override public Observable<House>
call(Community community) { return
observable.from(community.getHouses()); }
}) .last(new Func1<House, Boolean>() {
@Override public Boolean call(House house)
{
return "马德里春
天".equals(house.getCommunityName()); }
}) .subscribe(new Action1<House>() {
@Override public void call(House house) {
Log.e("rx_test", "last: " +
house.getCommunityName() + "小区，大小: " +
house.getSize()); } });
});
```

输出结果：

```
Last: 马德里春天小区，大小: 88.6
```

## 总结

到此，本篇关于RxJava的常用过滤类操作符就讲解完毕了，下一篇我们将一起研究RxJava的四类操作符中的组合操作符都有哪些以及如何使用。

技术渣一枚，有写的不对的地方欢迎大神们留言指正，有什么疑惑或者建议也可以在我Github上RxJavaDemo项目Issues中提出，我会及时回复。

附上RxJavaDemo的地址：

[RxJavaDemo](#)

## 19.4 RxJava操作符之组合操作符(四)

### 前言

上一篇文章我们学习了过滤类操作符，本篇我们将一起来学习RxJava组合类操作符。组合操作符主要是用来同时处理多个Observable，将他们进行组合创建出新的满足我们需求的Observable，一起来看下都有哪些。

## 组合操作符

### Merge

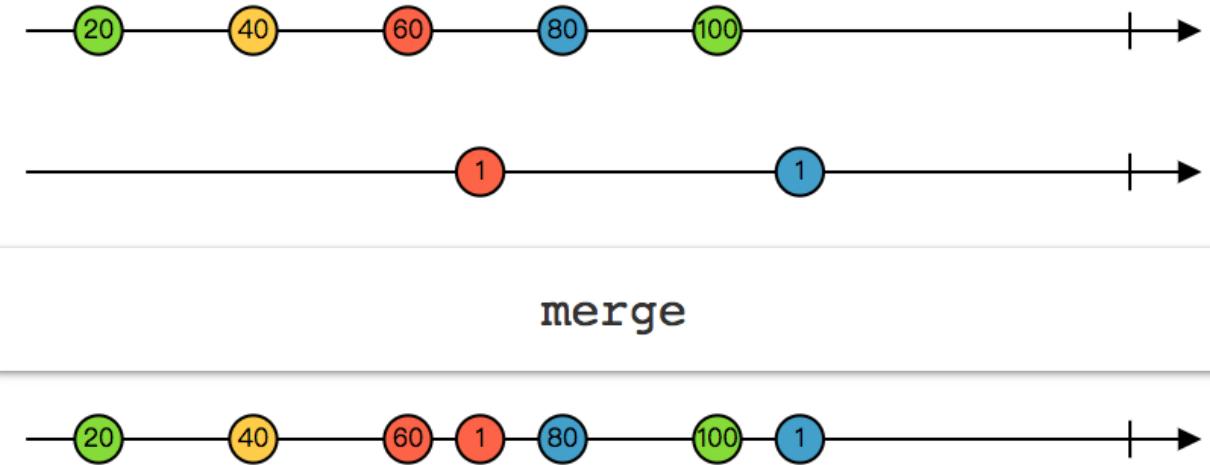
merge操作符，将两个Observable要发射的观测序列合并为一个序列进行发射。按照两个序列每个元素的发射时间先后进行排序，同一时间点发射的元素则是无序的。

```
//将一个发送字母的Observable与发送数字的Observable合并发射final
String[] words = new String[]{"A", "B", "C", "D", "E",
 "F", "G", "H", "I"};//字母Observable, 每200ms发射一次
observable<String> wordSequence =
observable.interval(200, TimeUnit.MILLISECONDS)
.map(new Func1<Long, String>() { @Override
 public String call(Long position) {
 return words[position.intValue()]; } })
 .take(words.length);//数字Observable, 每500ms发射一次
observable<Long> numberSequence =
observable.interval(500,
TimeUnit.MILLISECONDS).take(4); observable.merge(wordSequence,
numberSequence).subscribe(new
Action1<Serializable>() { @Override
public void call(Serializable serializable) {
 Log.e("rx_test", "merge: " + serializable.toString());
} });
});
```

输出结果：

```
merge: Amerge: Bmerge: Omerge: Cmerge: Dmerge: Emerge:
1merge: Fmerge: Gmerge: 2merge: Hmerge: Imerge: 3
```

原理图：



`merge`操作符还有一种入参`merge(Observable[])`,可传入含有多个`Observable`的集合, `merge`操作符也可将这多个`Observable`的序列合并后发射。

### MergeDelayError

`mergeDelayError`操作符, 与`merge`功能类似, 都是用来合并`Observable`的。不同之处在于`mergeDelayError`操作符在合并过程中发生异常的话不会立即停止合并, 而会在所有元素合并发射完毕之后再发射异常。但发生异常的那个`Observable`就不会发射数据了。

```

//字母observable, 每200ms发射一次, 模拟过程中产生一个异常
observable<String> wordSequence =
observable.interval(200, TimeUnit.MILLISECONDS)
.map(new Func1<Long, String>() { @Override
 public String call(Long position) {
 Long cache = position; if (cache == 3) {
 cache = cache / 0; }
 return words[position.intValue()];
 } }) .take(words.length); //数字observable, 每
500ms发射一次
observable<Long> numberSequence =
observable.interval(500,
TimeUnit.MILLISECONDS).take(4); observable.mergeDelayError
(wordSequence, numberSequence) .subscribe(new
Action1<Serializable>() { @Override
public void call(Serializable serializable) {
 Log.e("rx_test", "mergeDelayError: " +
serializable.toString()); } }, new
Action1<Throwable>() { @Override
public void call(Throwable throwable) {
 Log.e("rx_test", "mergeDelayError: " +
throwable.getMessage()); } }, new
Action0() { @Override
call() { Log.e("rx_test",
"mergeDelayError: onComplete"); } });

```

输出结果：

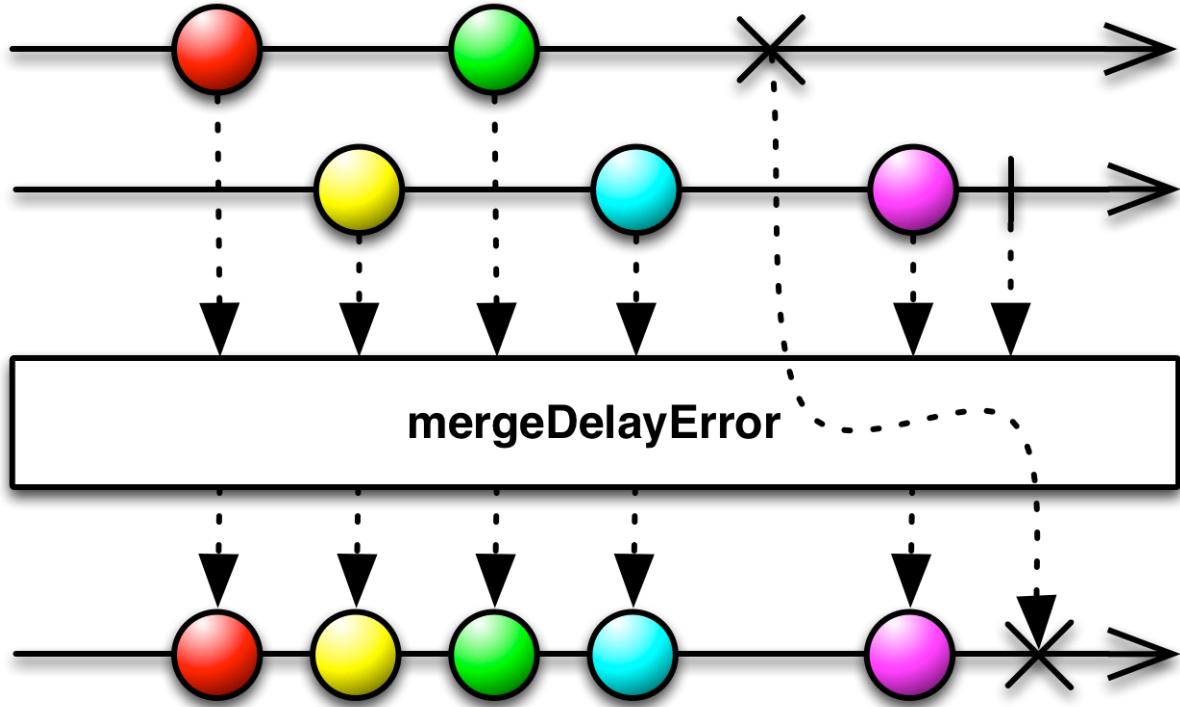
```

mergeDelayError: AmergeDelayError: BmergeDelayError:
OmergeDelayError: CmergeDelayError: 1mergeDelayError:
2mergeDelayError: 3mergeDelayError: divide by zero

```

由输出结果可看出，wordSequence在发射到C时抛出了一个异常，停止发射其剩下的数据，但合并没有停止。合并完成之后这个异常才被发射了出来。

原理图：



## Concat

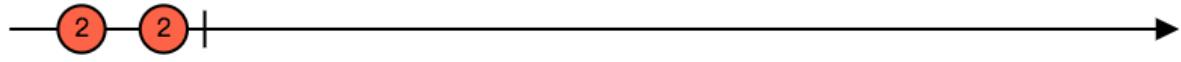
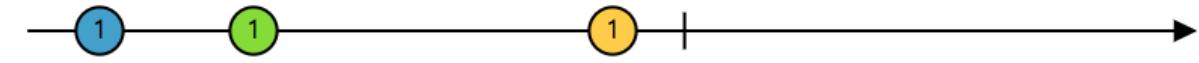
concat操作符，将多个Observable发射的数据进行合并后发射，类似于merge操作符。但concat操作符是将Observable依次发射，是有序的。

```
observable<String> wordSequence = observable.just("A",
 "B", "C", "D", "E"); observable<Integer> numberSequence =
observable.just(1, 2, 3, 4, 5); observable<String>
nameSequence = observable.just("Sherlock", "Holmes",
 "Xu", "Lei"); observable.concat(wordSequence,
 numberSequence, nameSequence) .subscribe(new
Action1<Serializable>() {
 @Override
 public void call(Serializable serializable) {
 Log.e("rx_test", "concat: " + serializable.toString());
 }
});
```

输出结果：

```
concat: Aconcat: Bconcat: Cconcat: Dconcat: Econcat:
1concat: 2concat: 3concat: 4concat: 5concat: Sherlockconcat:
Holmesconcat: Xuconcat: Lei
```

原理图：



## concat



## Zip

zip(Observable, Observable, Func2)操作符，根据Func2中的call()方法规则合并两个Observable的数据项并发射。

**注意：若其中一个Observable数据发送结束或出现异常后，另一个Observable也会停止发射数据。**

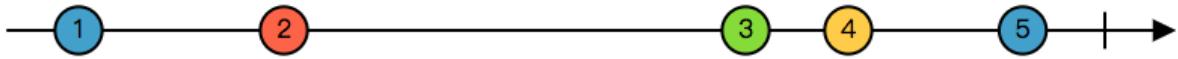
```
Observable<String> wordSequence = Observable.just("A",
 "B", "C", "D", "E"); Observable<Integer> numberSequence =
 Observable.just(1, 2, 3, 4, 5,
 6); Observable.zip(wordSequence, numberSequence, new
 Func2<String, Integer, String>() { @Override public
 String call(String s, Integer integer) { return s
 + integer; } }).subscribe(new Action1<String>() {
 @Override public void call(String s) {
 Log.e("rx_test", "zip: " + s); } });
```

输出结果：

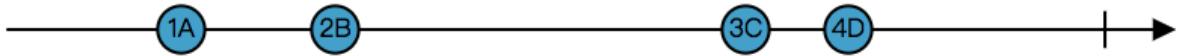
```
zip: A1zip: B2zip: C3zip: D4zip: E5
```

由输出结果可看出numberSequence观测序列最后的6并没有发射出来，由于wordSequence观测序列已发射完所有数据，所以组合序列也停止发射数据了。

原理图：



**zip**



## StartWith

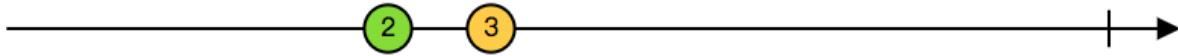
startWith操作符，用于在源Observable发射的数据前，插入指定的数据并发射。

```
observable.just(4, 5, 6, 7) .startwith(1, 2, 3)
 .subscribe(new Action1<Integer>() {
@Override public void call(Integer integer) {
 Log.e("rx_test", "startwith: " + integer);
 });
});
```

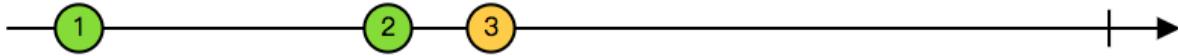
输出结果：

```
startwith: 1startwith: 2startwith: 3startwith: 4startwith:
5startwith: 6startwith: 7
```

原理图：



**startWith(1)**



startWith还有两种入参：

- `startWith(Iterable)`: 可在源Observable发射的数据前插入Iterable数据并发射。
- `startWith(Observable)`: 可在源Observable发射的数据前插入另一 Observable发射的数据并发射。

## SwitchOnNext

`switchOnNext`操作符，用来将一个发射多个小Observable的源 Observable转化为一个Observable，然后发射多个小Observable所发射的数据。若小Observable正在发射数据时，源Observable又发射了新的小Observable，则前一个小Observable还未发射的数据会被抛弃，直接发射新的小Observable所发射的数据，上例子。

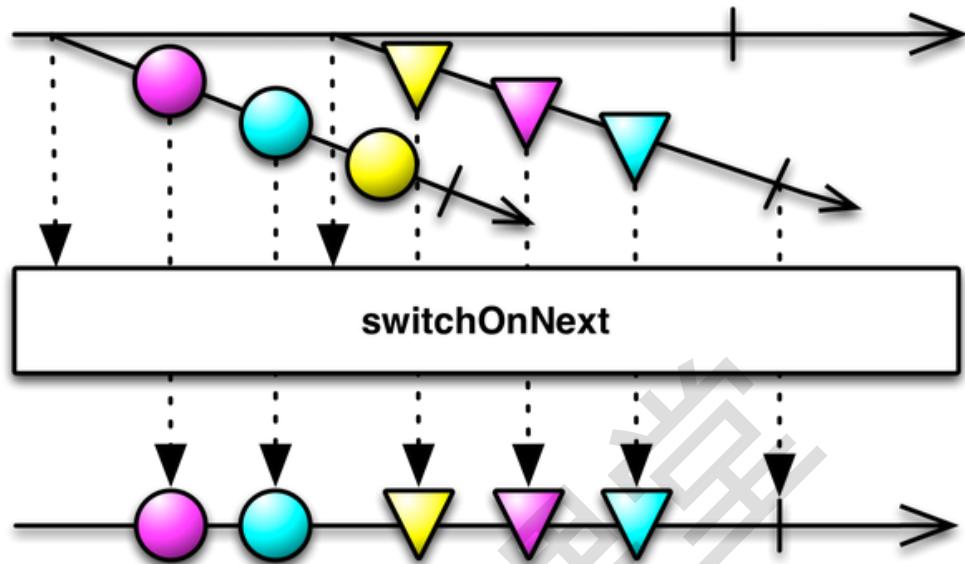
```
//每隔500ms产生一个Observable<Observable<Long>>
observable = Observable.interval(500,
 TimeUnit.MILLISECONDS)
 .map(new Func1<Long, Observable<Long>>() {
 @Override
 public Observable<Long> call(Long aLong) {
 //每隔200毫秒产生一组数据 (0,10,20,30,40)
 return Observable.interval(200, TimeUnit.MILLISECONDS)
 .map(new Func1<Long, Long>() {
 @Override
 public Long call(Long aLong) {
 return aLong * 10;
 }
 }).take(5);
 }
 }).take(2);
 observable.switchOnNext(observable)
 .subscribe(new Action1<Long>() {
 @Override
 public void call(Long aLong) {
 Log.e("rx_test", "switchOnNext: " + aLong);
 }
 });
}
```

输出结果：

```
switchOnNext: 0switchOnNext: 10switchOnNext:
0switchOnNext: 10switchOnNext: 20switchOnNext:
30switchOnNext: 40
```

由输出结果发现第一个小Observable打印到10则停止了发射数据，说明其发射到10时，新的小Observable被创建了出来，第一个小Observable则被中断发射，开始发射新的小Observable的数据。

原理图：



### CombineLatest

`combineLatest`操作符，用于将两个Observale最近发射的数据以Func2函数的规则进行组合并发射。

```
//引用merge的例子final String[] words = new String[]{"A",
"B", "C", "D", "E", "F", "G", "H",
"I"};Observable<String> wordSequence =
Observable.interval(300, TimeUnit.MILLISECONDS)
.map(new Func1<Long, String>() {
 @Override
 public String call(Long position) {
 return words[position.intValue()];
 }
}).take(words.length);Observable<Long>
numberSequence = Observable.interval(500,
TimeUnit.MILLISECONDS)
.take(5);Observable.combineLatest(wordSequence,
numberSequence, new Func2<String, Long, String>()
{
 @Override
 public String call(String s, Long aLong) {
 return s +
aLong; }
}.subscribe(new
Action1<Serializable>()
{
 @Override
 public void call(Serializable serializable) {
 Log.e("rx_test", "combineLatest: " +
serializable.toString()); }
});
```

输出结果：

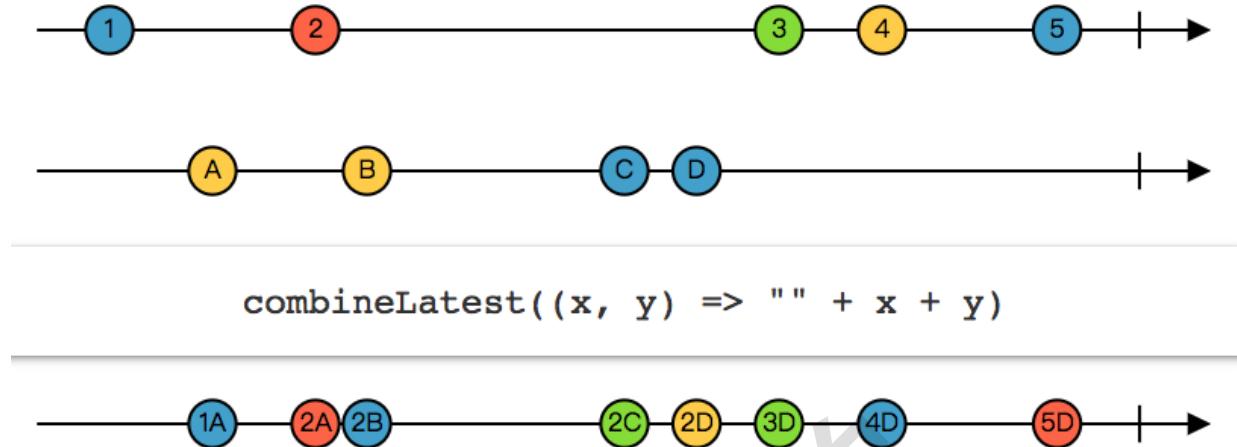
```
combineLatest: A0combineLatest: B0combineLatest:
C0combineLatest: C1combineLatest: D1combineLatest:
E1combineLatest: E2combineLatest: F2combineLatest:
F3combineLatest: G3combineLatest: H3combineLatest:
H4combineLatest: I4
```

如果将wordSequence与numberSequence的入参顺序互换，输出结果也会不同：

```
combineLatest: 0AcombineLatest: 0BcombineLatest:
0CcombineLatest: 1CcombineLatest: 1DcombineLatest:
2DcombineLatest: 2EcombineLatest: 2FcombineLatest:
3FcombineLatest: 3GcombineLatest: 3HcombineLatest:
4HcombineLatest: 4I
```

wordSequence每300ms发射一个字符，numberSequence每500ms发射一个数字。可能有些码友不知道这个输出结果怎么来的，这个操作符确实不太好理解。我们来看一下这个原理图就很清楚了。

原理图：



## Join

join(Observable, Func1, Func1, Func2)操作符，类似于combineLatest操作符，用于将ObservableA与ObservableB发射的数据进行排列组合。但join操作符可以控制Observable发射的每个数据的生命周期，在每个发射数据的生命周期内，可与另一个Observable发射的数据按照一定规则进行合并，来看下join的几个入参。

- Observable：需要与源Observable进行组合的目标Observable。
- Func1：接收从源Observable发射来的数据，并返回一个Observable，这个Observable的声明周期决定了源Observable发射出来的数据的有效期；
- Func1：接收目标Observable发射来的数据，并返回一个Observable，这个Observable的声明周期决定了目标Observable发射出来的数据的有效期；
- Func2：接收从源Observable和目标Observable发射出来的数据，并将这两个数据按自定的规则组合后返回。

```

//产生字母的序列,周期为1000ms
String[] words = new String[]
{"A", "B", "C", "D", "E", "F", "G",
"H"};
Observable<String> observableA =
Observable.interval(1000, TimeUnit.MILLISECONDS)
.map(new Func1<Long, String>() {
 @Override
 public String call(Long aLong) {
 return words[aLong.intValue()];
 }
}).take(8); //产0,1,2,3,4,5,6,7的序列,延时500ms发射,周期为
1000ms
Observable<Long> observableB =
Observable.interval(500, 1000, TimeUnit.MILLISECONDS)
.map(new Func1<Long, Long>() {
 @Override
 public Long call(Long aLong) {
 return aLong;
 }
}).take(words.length); //join
observableA.join(observableB,
new Func1<String, Observable<Long>>() {
 @Override
 public Observable<Long> call(String s) {
 //ObservableA发射的数据有效期为600ms
 return Observable.timer(600,
TimeUnit.MILLISECONDS);
 }
}, new Func1<Long, Observable<Long>>() {
 @Override
 public Observable<Long> call(Long aLong) {
 //ObservableB发射的数据有效期为600ms
 return Observable.timer(600, TimeUnit.MILLISECONDS);
 }
}, new Func2<String, Long, String>()
{
 @Override
 public String call(String s, Long aLong) {
 return s +
aLong;
 }
}).subscribe(new
Action1<String>() {
 @Override
 public void call(String s) {
 Log.e("rx_test", "join: " + s);
 }
});

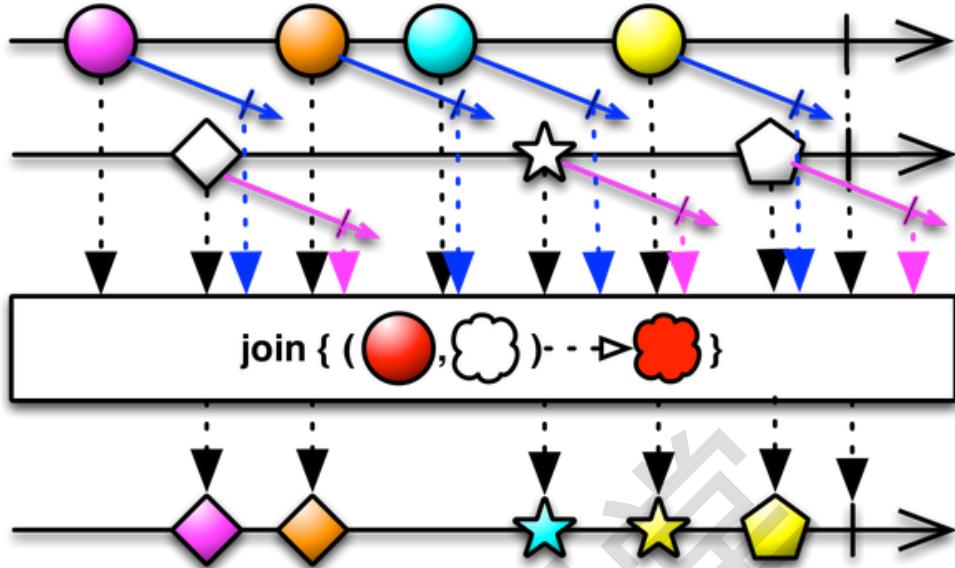
```

join操作符的组合方式类似于数学上的排列组合规则，以ObservableA为基准源Observable，按照其自身周期发射数据，且每个发射出来的数据都有其有效期。而ObservableB每发射出来一个数据，都与A发射出来的并且还在有效期内的数据按Func2函数中的规则进行组合，B发射出来的数据也有其有效期。最后再将结果发射给观察者进行处理。

输出结果：

```
join: A0join: A1join: B1join: B2join: C2join: C3join:
D3join: D4join: E4join: E5join: F5join: F6join: G6join:
G7join: H7
```

原理图：



## GroupJoin

groupJoin操作符，类似于join操作符，区别在于第四个参数Func2的传入函数不同，对join之后的结果包装了一层小的Observable，便于用户再次进行一些过滤转换等操作再发射给Observable。

```

observableA.groupJoin(observableB, new
 Func1<String, Observable<Long>>() { @Override
 public Observable<Long> call(String s) {
 return Observable.timer(600,
 TimeUnit.MILLISECONDS); } }, new
 Func1<Long, Observable<Long>>() { @Override
 public Observable<Long> call(Long aLong) {
 return Observable.timer(600,
 TimeUnit.MILLISECONDS); } }, new
 Func2<String, Observable<Long>, Observable<String>>() {
 @Override public Observable<String>
call(final String s, Observable<Long> longObservable) {
 return longObservable.map(new Func1<Long,
 String>() { @Override
 public String call(Long aLong) {
 return s + aLong; } });
 } });
 .subscribe(new
 Action1<Observable<String>>() { @Override
 public void call(Observable<String>
stringObservable) {
 stringObservable.subscribe(new Action1<String>() {
 @Override public void
Log.e("rx_test",
 "groupJoin: " + s); });
 } });
 });
}

```

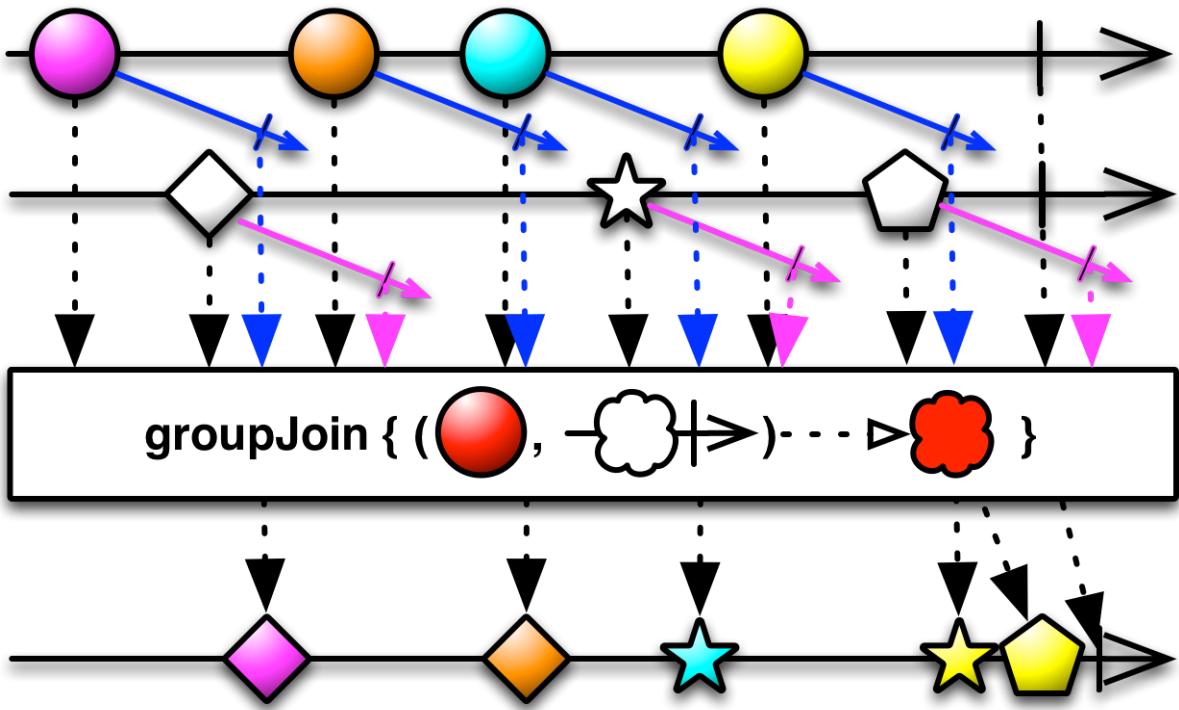
输出结果：

```

groupJoin: A0groupJoin: A1groupJoin: B1groupJoin:
B2groupJoin: C2groupJoin: C3groupJoin: D3groupJoin:
D4groupJoin: E4groupJoin: E5groupJoin: F5groupJoin:
F6groupJoin: G6groupJoin: G7groupJoin: H7

```

原理图：



## 总结

到此，本篇关于RxJava的常用组合类操作符就讲解完毕了。通过以上四篇文章对RxJava四类操作符的学习，相信大家已经基本掌握RxJava如何使用了。实践是检验真理的唯一标准，下一篇我们来一起上项目看看实践中如何使用RxJava。

技术渣一枚，有写的不对的地方欢迎大神们留言指正，有什么疑惑或者建议也可以在我Github上RxJavaDemo项目Issues中提出，我会及时回复。

附上RxJavaDemo的地址：

[RxJavaDemo](#)

# 第二十节 RxJava原理与源码分析

## 20.1 RxJava的消息订阅和线程切换原理

### 20.1.1 前言

本文主要是对RxJava的消息订阅和线程切换进行源码分析，相关的使用方式等不作详细介绍。

本文源码基于 rxjava:2.1.14。

## 20.1.2 RxJava简介

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

It extends the observer pattern to support sequences of data/events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures.

上面这段话来自于RxJava在github上面的官方介绍。翻译成中文的大概意思就是：

RxJava是一个在Java虚拟机上的响应式扩展，通过使用可观察的序列将异步和基于事件的程序组合起来的一个库。

它扩展了观察者模式来支持数据/事件序列，并且添加了操作符，这些操作符允许你声明性地组合序列，同时抽象出要关注的问题：比如低级线程、同步、线程安全和并发数据结构等。

简单点来说， RxJava就是一个使用了观察者模式，能够异步的库。

## 20.1.3 观察者模式

上面说到， RxJava扩展了观察者模式，那么什么是观察模式呢？我们先来了解一下。

举个例子，以微信公众号为例，一个微信公众号会不断产生新的内容，如果我们读者对这个微信公众号的内容感兴趣，就会订阅这个公众号，当公众号有新内容时，就会推送给我们的。我们收到新内容时，如果是我们感兴趣的，就会点进去看下;如果是广告的话，就可能直接忽略掉。这就是我们生活中遇到的典型的观察者模式。

在上面的例子中，微信公众号就是一个被观察者(observable)，不断的产生内容（事件），而我们读者就是一个观察者(observer），通过订阅(subscribe) 就能够接受到微信公众号（被观察者）推送的内容（事件），根据不同的内容（事件）做出不同的操作。

### 3.1 Rxjava角色说明

RxJava的扩展观察者模式中就是存在这么4种角色：

| 角色                               | 角色功能          |
|----------------------------------|---------------|
| 被观察者 ( <code>Observable</code> ) | 产生事件          |
| 观察者 ( <code>Observer</code> )    | 响应事件并做出处理     |
| 事件 ( <code>Event</code> )        | 被观察者和观察者的消息载体 |
| 订阅 ( <code>Subscribe</code> )    | 连接被观察者和观察者    |

### 3.2 RxJava事件类型

RxJava中的事件分为三种类型：`Next`事件、`Complete`事件和`Error`事件。具体如下：

| 事件类型                  | 含义   | 说明                                                                                  |
|-----------------------|------|-------------------------------------------------------------------------------------|
| <code>Next</code>     | 常规事件 | 被观察者可以发送无数个 <code>Next</code> 事件，观察者也可以接受无数个 <code>Next</code> 事件                   |
| <code>Complete</code> | 结束事件 | 被观察者发送 <code>Complete</code> 事件后可以继续发送事件，观察者收到 <code>Complete</code> 事件后将不会接受其他任何事件 |
| <code>Error</code>    | 异常事件 | 被观察者发送 <code>Error</code> 事件后，其他事件将被终止发送，观察者收到 <code>Error</code> 事件后将不会接受其他任何事件    |

## 20.1.4 RxJava的消息订阅

在分析RxJava消息订阅原理前，我们还是先来看下它的简单使用步骤。这里为了方便讲解，就不用链式代码来举例了，而是采用分步骤的方式来逐一说明（平时写代码的话还是建议使用链式代码来调用，因为更加简洁）。其使用步骤如下：

1. 创建被观察者(`Observable`)，定义要发送的事件。
2. 创建观察者(`Observer`)，接受事件并做出响应操作。
3. 观察者通过订阅 (`subscribe`) 被观察者把它们连接到一起。

### 4.1 RxJava的消息订阅例子

这里我们就根据上面的步骤来实现这个例子，如下：

```
//步骤1. 创建被观察者(Observable)，定义要发送的事件。
Observable observable = Observable.create(new
ObservableOnSubscribe<String>() {
 @Override
 public void subscribe(ObservableEmitter<String>
 emitter) throws Exception {
 emitter.onNext("文章1");
 emitter.onNext("文章2");
 emitter.onNext("文章3");
 emitter.onComplete();
 }
});

//步骤2. 创建观察者(Observer)，接受事件并做出响应操作。
Observer<String> observer = new Observer<String>() {
 @Override
 public void onSubscribe(Disposable d) {
 Log.d(TAG, "onSubscribe");
 }
 @Override
 public void onNext(String s) {
 Log.d(TAG, "onNext : " + s);
 }
 @Override
 public void onError(Throwable e) {
 Log.d(TAG, "onError : " + e.toString());
 }
 @Override
 public void onComplete() {
 Log.d(TAG, "onComplete");
 }
};

//步骤3. 观察者通过订阅(subscribe) 被观察者把它们连接到一起。
observable.subscribe(observer);
```

其输出结果为：

```
onSubscribeonNext : 文章1onNext : 文章2onNext : 文章
3onComplete
```

## 4.2 源码分析

下面我们将对消息订阅过程中的源码进行分析，分为两部分：创建被观察者过程和订阅过程。

### 4.2.1 创建被观察者过程

首先来看下创建被观察者(Observable)的过程，上面的例子中我们是直接使用`Observable.create()`来创建`Observable`，我们点进去这个方法看下。

#### 4.2.1.1 Observable类的create()

```
public static <T> Observable<T>
create(ObservableOnSubscribe<T> source) {
ObjectHelper.requireNonNull(source, "source is null");
 return RxJavaPlugins.onAssembly(new
ObservableCreate<T>(source)); }
```

可以看到，`create()`方法中也没做什么，就是创建一个`ObservableCreate`对象出来，然后把我们自定义的`ObservableOnSubscribe`作为参数传到`ObservableCreate`中去，最后就是调用`RxJavaPlugins.onAssembly()`方法。

我们先来看看`ObservableCreate`类：

#### 4.2.1.2 ObservableCreate类

```
public final class ObservableCreate<T> extends
Observable<T> {//继承自Observable public
ObservableCreate(ObservableOnSubscribe<T> source) {
this.source = source;//把我们创建的ObservableOnSubscribe对象
赋值给source。 }}
```

可以看到，`ObservableCreate`是继承自`Observable`的，并且会把`ObservableOnSubscribe`对象给存起来。

再看下 `RxJavaPlugins.onAssembly()` 方法

#### 4.2.1.3 RxJavaPlugins类的onAssembly()

```
public static <T> Observable<T> onAssembly(@NotNull
Observable<T> source) { //省略无关代码
 return source;
}
```

很简单，就是把上面创建的 `observableCreate` 给返回。

#### 4.2.1.4 简单总结

所以 `observable.create()` 中就是把我们自定义的 `ObservableOnSubscribe` 对象重新包装成一个 `observableCreate` 对象，然后返回这个 `observableCreate` 对象。

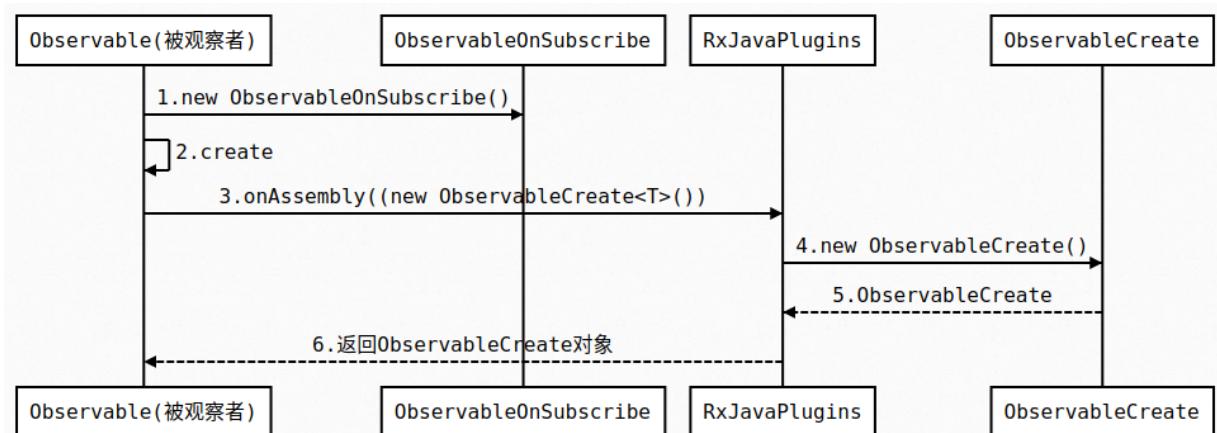
注意，这种重新包装新对象的用法在 RxJava 中会频繁用到，后面的分析中我们还会多次遇到。

放个图好理解，包起来哈 ~



#### 4.2.1.5 时序图

`observable.create()` 的时序图如下所示：



## 4.2.2 订阅过程

接下来我们就看下订阅过程的代码，同样，点进去  
`observable.subscribe()`：

### 4.2.2.1 Observable类的subscribe()

```
public final void subscribe(Observer<? super T>
observer) { //省略无关代码
 observer = RxJavaPlugins.onSubscribe(this, observer);
 subscribeActual(observer);
//省略无关代码 }
```

可以看到，实际上其核心的代码也就两句，我们分开来看下：

### 4.2.2.2 RxJavaPlugins类的onSubscribe()

```
public static <T> Observer<? super T>
onSubscribe(@NonNull Observable<T> source, @NonNull
Observer<? super T> observer) { //省略无关代码
 return observer; }
```

跟之前代码一样，这里同样也是把原来的`observer`返回而已。  
再来看下`subscribeActual()`方法。

### 4.2.2.3 Observable类的subscribeActual()

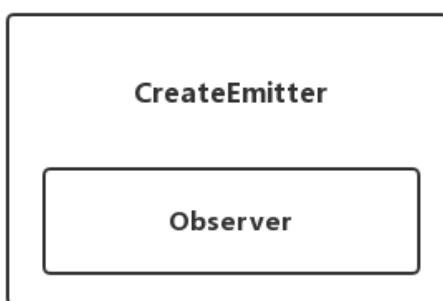
```
protected abstract void subscribeActual(Observer<?
super T> observer);
```

`observable`类的`subscribeActual()`中的方法是一个抽象方法，那么其具体实现在哪呢？还记得我们前面创建被观察者的过程吗，最终会返回一个`ObservableCreate`对象，这个`ObservableCreate`就是`Observable`的子类，我们点进去看下：

### 4.2.2.4 ObservableCreate类的subscribeActual()

```
 @Override protected void
subscribeActual(Observer<? super T> observer) {
CreateEmitter<T> parent = new CreateEmitter<T>(observer);
 //触发我们自定义的Observer的onSubscribe(Disposable)方法
 observer.onSubscribe(parent); try {
 source.subscribe(parent); } catch (Throwable ex) {
 Exceptions.throwIfFatal(ex);
parent.onError(ex); } }
```

可以看到，`subscribeActual()`方法中首先会创建一个`CreateEmitter`对象，然后把我们自定义的观察者`observer`作为参数给传进去。这里同样也是包装起来，放个图：



这个`CreateEmitter`实现了`ObservableEmitter`接口和`Disposable`接口，如下：

```
static final class CreateEmitter<T> extends
AtomicReference<Disposable> implements
ObservableEmitter<T>, Disposable { //代码省略 }
```

然后就是调用了`observer.onSubscribe(parent)`，实际上就是调用观察者的`onSubscribe()`方法，即告诉观察者已经成功订阅到了被观察者。

继续往下看，`subscribeActual()`方法中会继续调用`source.subscribe(parent)`，这里的`source`就是`ObservableOnSubscribe`对象，即这里会调用`ObservableOnSubscribe`的`subscribe()`方法。我们具体定义的`subscribe()`方法如下：

```
Observable observable = Observable.create(new
ObservableOnSubscribe<String>() { @Override
 public void subscribe(ObservableEmitter<String>
emitter) throws Exception {
 emitter.onNext("文章1"); emitter.onNext("文
章2"); emitter.onNext("文章3");
 emitter.onComplete(); } });
});
```

`ObservableEmitter`, 顾名思义, 就是被观察者发射器。

所以, `subscribe()`里面的三个`onNext()`方法和一个`onComplete()`会逐一被调用。

这里的`ObservableEmitter`接口其具体实现为`CreateEmitter`, 我们看看`CreateEmitter`类的`onNext()`方法和`onComplete()`的实现:

#### 4.2.2.5 CreateEmitter类的onNext()和onComplete()等

```
//省略其他代码 @Override
public void onNext(T t) { //省略无关代码
 if (!isDisposed()) { //调用观察者的
 onNext(); observer.onNext(t); }
 }
@Override public void
onComplete() { if (!isDisposed()) {
 //调用观察者的onComplete()
 onComplete(); }
finally { dispose(); }
}
}
```

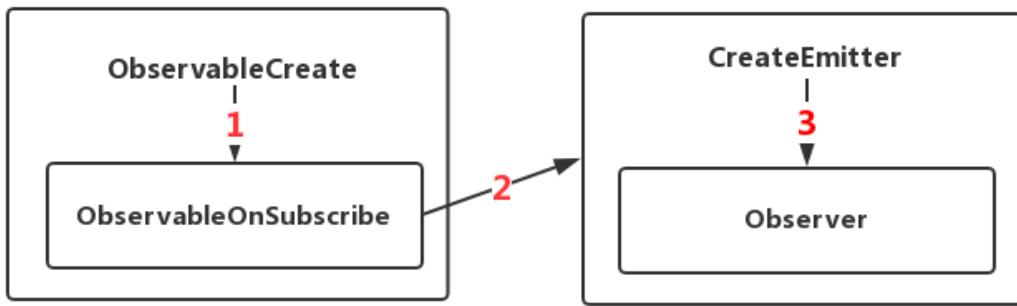
可以看到, 最终就是会调用到观察者的`onNext()`和`onComplete()`方法。至此, 一个完整的消息订阅流程就完成了。

另外, 可以看到, 上面有个`isDisposed()`方法能控制消息的走向, 即能够切断消息的传递, 这个后面再来说。

#### 4.2.2.6 简单总结

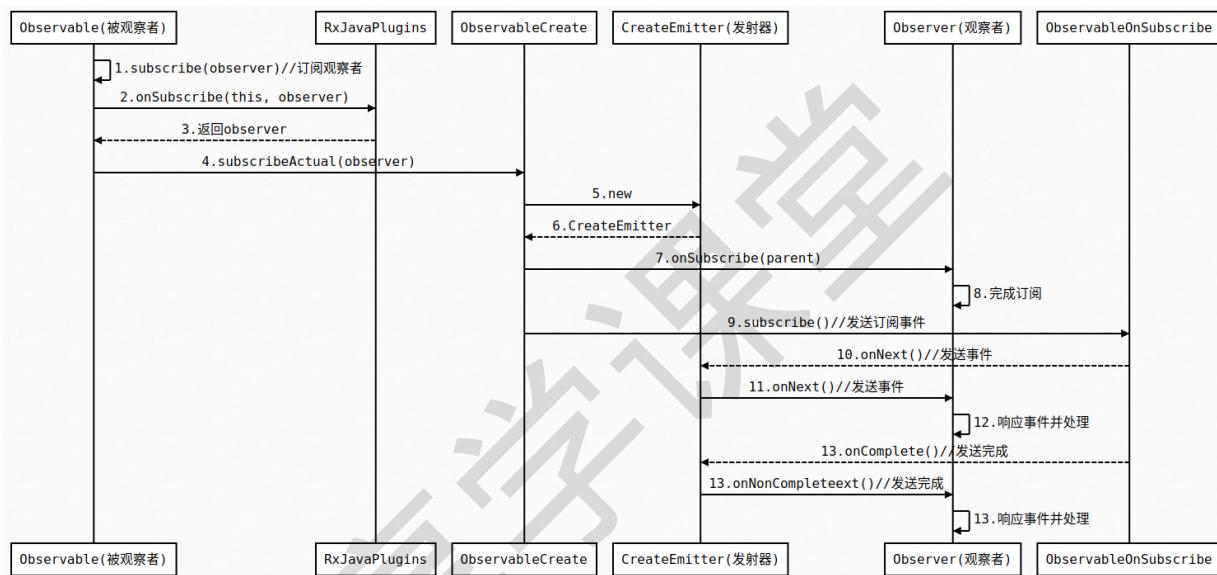
`Observable`(被观察者)和`Observer`(观察者)建立连接(订阅)之后, 会创建出一个发射器`CreateEmitter`, 发射器会把被观察者中产生的事件发送到观察者中去, 观察者对发射器中发出的事件做出响应处理。可以看到, 是订阅之后, `Observable`(被观察者)才会开始发送事件。

放张事件流的传递图：



#### 4.2.2.7 时序流程图

再来看下订阅过程的时序流程图：



#### 4.3 切断消息

之前有提到过切断消息的传递，我们先来看下如何使用：

##### 4.3.1 切断消息

```
Observable observable = Observable.create(new
ObservableOnSubscribe<String>() { @Override
 public void subscribe(ObservableEmitter<String>
emitter) throws Exception {
 emitter.onNext("文章1"); emitter.onNext("文
章2"); emitter.onNext("文章3");
 emitter.onComplete(); } });
 Observer<String> observer = new Observer<String>()
{
 private Disposable mDisposable;
 @Override public void onSubscribe(Disposable
d) { Log.d(TAG, "onSubscribe : " + d);
 mDisposable=d; }
 @Override public void onNext(String s) {
 Log.d(TAG, "onNext : " + s);
 mDisposable.dispose(); Log.d(TAG, "切断观察
者与被观察者的连接"); }
 @Override public void onError(Throwable e) {
 Log.d(TAG, "onError : " + e.toString()); }
 @Override public void onComplete() {
 Log.d(TAG, "onComplete"); }
}; observable.subscribe(observer);
```

输出结果为：

```
onSubscribe : nullonNext : 文章1切断观察者与被观察者的连接
```

可以看到，要切断消息的传递很简单，调用下`Disposable`的`dispose()`方法即可。调用`dispose()`之后，被观察者虽然能继续发送消息，但是观察者却收不到消息了。

另外有一点需要注意，上面`onSubscribe`输出的`Disposable`值是“`null`”，并不是空引用`null`。

### 4.3.2 切断消息源码分析

我们这里来看看下`dispose()`的实现。`Disposable`是一个接口，可以理解`Disposable`为一个连接器，调用`dispose()`后，这个连接器将会中断。其具体实现在`CreateEmitter`类，之前也有提到过。我们来看下`CreateEmitter`的`dispose()`方法：

#### 4.3.2.1 CreateEmitter的dispose()

```
@Override public void dispose() {
 DisposableHelper.dispose(this); }
```

就是调用DisposableHelper.dispose(this)而已。

#### 4.3.2.2 DisposableHelper类

```
public enum DisposableHelper implements Disposable {
 DISPOSED; //其他代码省略
 public static boolean isDisposed(Disposable d) { //判断Disposable
 //类型的变量的引用是否等于DISPOSED //即判断该连接器是否被中
 return d == DISPOSED; }
 public static boolean dispose(AtomicReference<Disposable> field) {
 Disposable current = field.get();
 Disposable d = DISPOSED;
 if (current != d) {
 current = field.getAndSet(d);
 if (current != d) {
 if (current != null) {
 current.dispose();
 }
 }
 }
 return true; }
 return false; }}
```

可以看到DisposableHelper是一个枚举类，并且只有一个值:DISPOSED。dispose()方法中会把一个原子引用field设为DISPOSED，即标记为中断状态。因此后面通过isDisposed()方法即可判断连接器是否被中断。

#### 4.3.2.3 CreateEmitter类中的方法

再回头看看CreateEmitter类中的方法：

```

@Override public void onNext(T t) {
 //省略无关代码
 if (!isDisposed()) { //如果没有dispose(), 才会调用onNext()
 observer.onNext(t);
 }
}

@Override public void onError(Throwable t) {
 if (!tryOnError(t)) {
 //如果dispose()了, 会调用到这里, 即最终会崩溃
 RxJavaPlugins.onError(t);
 }
}

@Override public boolean tryOnError(Throwable t) {
 //省略无关代码
 if (!isDisposed()) {
 try {
 //如果没有dispose(), 才会调用onError()
 observer.onError(t);
 } finally {
 //onError()之后会dispose()
 dispose();
 }
 }
 return true;
}

//如果dispose()了, 返回false
return false;
}

@Override public void onComplete() {
 if (!isDisposed()) {
 //如果没有dispose(), 才会调用onComplete()
 observer.onComplete();
 }
}

```

从上面的代码可以看到：

1. 如果没有 `dispose`, `observer.onNext()` 才会被调用到。
2. `onError()` 和 `onComplete()` 互斥, 只能其中一个被调用到, 因为调用了他们的任意一个之后都会调用 `dispose()`。
3. 先 `onError()` 后 `onComplete()`, `onComplete()` 不会被调用到。

反过来, 则会崩溃, 因为 `onError()` 中抛出了异常:

`RxJavaPlugins.onError(t)`。实际上是 `dispose` 后继续调用 `onError()` 都会炸。

## 20.1.5 RxJava的线程切换

上面的例子和分析都是在同一个线程中进行，这中间也没涉及到线程切换的相关问题。但是在实际开发中，我们通常需要在一个子线程中去进行一些数据获取操作，然后要在主线程中去更新UI，这就涉及到线程切换的问题了，通过RxJava我们也可以把线程切换写得还简洁。

### 5.1 线程切换例子

关于RxJava如何使用线程切换，这里就不详细讲了。

我们直接来看一个例子，并分别打印RxJava在运行过程中各个角色所在的线程。

```
 new Thread() { @Override
public void run() { Log.d(TAG, "Thread
run() 所在线程为 :" + Thread.currentThread().getName());
 Observable
.create(new ObservableOnSubscribe<String>() {
 @Override
public void subscribe(ObservableEmitter<String> emitter)
throws Exception {
Log.d(TAG, "Observable subscribe() 所在线程为 :" +
Thread.currentThread().getName());
 emitter.onNext("文章1");
 emitter.onNext("文章2");
 emitter.onComplete();
 })
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(new Observer<String>() {
 @Override
public void onSubscribe(Disposable d) {
Log.d(TAG, "Observer onSubscribe() 所在线程为
:" + Thread.currentThread().getName());
 } @Override
 public void onNext(String s) {
Log.d(TAG, "Observer onNext() 所在线程
为 :" + Thread.currentThread().getName());
 } @Override
 public void onError(Throwable e) {
Log.d(TAG, "Observer onError()
所在线程为 :" + Thread.currentThread().getName());
 } @Override
 public void onComplete() {
Log.d(TAG, "Observer
onComplete() 所在线程为 :" +
Thread.currentThread().getName());
 }); }
}).start();
```

输出结果为：

```
Thread run() 所在线程为 :Thread-2
observer onSubscribe() 所在线程为
:Thread-2
observable subscribe() 所在线程为
:RxCachedThreadsScheduler-1
observer onNext() 所在线程为
:mainObserver onNext() 所在线程为 :mainObserver
onComplete() 所在线程为 :main
```

从上面的例子可以看到：

1. `Observer` (观察者) 的 `onSubscribe()` 方法运行在当前线程中。
2. `Observable` (被观察者) 中的 `subscribe()` 运行在 `subscribeOn()` 指定的线程中。
3. `Observer` (观察者) 的 `onNext()` 和 `onComplete()` 等方法运行在 `observeOn()` 指定的线程中。

## 5.2 源码分析

下面我们对线程切换的源码进行一下分析，分为两部分：

`subscribeOn()` 和 `observeOn()`。

### 5.2.1 subscribeOn()源码分析

首先来看下 `subscribeOn()`，我们的例子中是这么个使用的：

```
.subscribeOn(Schedulers.io())
```

`subscribeOn()` 方法要传入一个 `Scheduler` 类对象作为参数，`Scheduler` 是一个调度类，能够延时或周期性地去执行一个任务。

#### 5.2.1.1 Scheduler类型

通过 `Schedulers` 类我们可以获取到各种 `Scheduler` 的子类。RxJava 提供了以下这些线程调度类供我们使用：

| Scheduler类型          | 使用方式                                        | 含义         | 使用场景                       |
|----------------------|---------------------------------------------|------------|----------------------------|
| IoScheduler          | <code>Schedulers.io()</code>                | io操作线程     | 读写SD卡文件，查询数据库，访问网络等IO密集型操作 |
| NewThreadScheduler   | <code>Schedulers.newThread()</code>         | 创建新线程      | 耗时操作等                      |
| SingleScheduler      | <code>Schedulers.single()</code>            | 单例线程       | 只需一个单例线程时                  |
| ComputationScheduler | <code>Schedulers.computation()</code>       | CPU计算操作线程  | 图片压缩取样、xml,json解析等CPU密集型计算 |
| TrampolineScheduler  | <code>Schedulers.trampoline()</code>        | 当前线程       | 需要在当前线程立即执行任务时             |
| HandlerScheduler     | <code>AndroidSchedulers.mainThread()</code> | Android主线程 | 更新UI等                      |

### 5.2.1.2 Schedulers类的io()

下面我们来看下 `schedulers.io()` 的代码，其他的 scheduler 子类都差不多，就不逐以分析了，有兴趣的请自行查看哈~

```

@NotNull static final Scheduler IO;
@NotNull public static Scheduler io() { //1.直接
 返回一个名为IO的Scheduler对象 return
 RxJavaPlugins.onIOScheduler(IO); } static {
 //省略无关代码 //2.IO对象是在静态代码块中实例
 //化的，这里会创建按一个IOTask() IO =
 RxJavaPlugins.initIOScheduler(new IOTask()); }
 static final class IOTask implements Callable<Scheduler>
 { @Override public Scheduler call() throws
 Exception { //3.IOTask中会返回一个IoHolder对象
 return IoHolder.DEFAULT; } }
 static final class IoHolder { //4.IoHolder中会就是
 new一个IoScheduler对象出来 static final Scheduler
 DEFAULT = new IoScheduler(); }

```

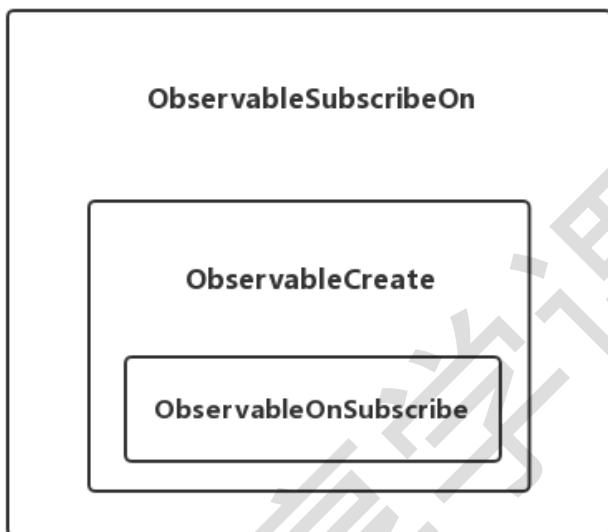
可以看到，`schedulers.io()` 中使用了静态内部类的方式来创建出了一个单例 `Ioscheduler` 对象出来，这个 `Ioscheduler` 是继承自 `Scheduler` 的。这里mark一发，后面会用到这个 `Ioscheduler` 的。

### 5.2.1.3 Observable类的subscribeOn()

然后，我们就来看下subscribeOn()的代码：

```
public final Observable<T> subscribeOn(Scheduler
scheduler) { //省略无关代码 return
RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>
(this, scheduler)); }
```

可以看到，首先会将当前的observable（其具体实现为observableCreate）包装成一个新的ObservableSubscribeOn对象。放个图：



跟前面一样，RxJavaPlugins.onAssembly()也是将ObservableSubscribeOn对象原样返回而已，这里就不看了。可以看下ObservableSubscribeOn的构造方法：

### 5.2.1.4 ObservableSubscribeOn类的构造方法

```
public ObservableSubscribeOn(ObservableSource<T>
source, Scheduler scheduler) { super(source);
this.scheduler = scheduler; }
```

也就是把source和scheduler这两个保存一下，后面会用到。

然后 `subscribeOn()` 方法就完了。好像也没做什么，就是重新包装一下对象而已，然后将新对象返回。即将一个旧的被观察者包装成一个新的被观察者。

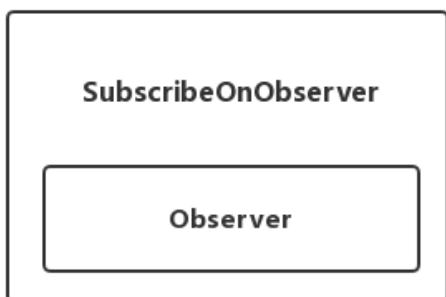
### 5.2.1.5 ObservableSubscribeOn类的subscribeActual()

接下来我们回到订阅过程，为什么要回到订阅过程呢？因为事件的发送是从订阅过程开始的啊。

虽然我们这里用到了线程切换，但是呢，其订阅过程前面的内容跟上一节分析的是一样的，我们这里就不重复了，直接从不一样的地方开始。还记得订阅过程中 `observable` 类的 `subscribeActual()` 是个抽象方法吗？因此要看其子类的具体实现。在上一节订阅过程中，其具体实现是在 `observableCreate` 类。但是由于我们调用 `subscribeOn()` 之后，`ObservableCreate` 对象被包装成了一个新的 `ObservableSubscribeOn` 对象了。因此我们就来看看 `ObservableSubscribeOn` 类中的 `subscribeActual()` 方法：

```
@Override public void subscribeActual(final Observer<? super T> s) { final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s); s.onSubscribe(parent); parent.setDisposable(scheduler.scheduleDirect(new SubscribeTask(parent))); }
```

`subscribeActual()` 中同样也将我们自定义的 `Observer` 给包装成了一个新的 `SubscribeOnObserver` 对象。同样，放张图：



然后就是调用 `Observer` 的 `onSubscribe()` 方法，可以看到，到目前为止，还没出现过任何线程相关的东西，所以 `Observer` 的 `onSubscribe()` 方法就是运行在当前线程中。

然后我们重点看下最后一行代码，首先创建一个 `SubscribeTask` 对象，然后就是调用 `scheduler.scheduleDirect()`。  
我们先来看下 `SubscribeTask` 类：

### 5.2.1.6 SubscribeTask类

```
//SubscribeTask是ObservableSubscribeOn的内部类 final
class SubscribeTask implements Runnable { private
final SubscribeOnObserver<T> parent;
SubscribeTask(SubscribeOnObserver<T> parent) {
this.parent = parent; } @Override
public void run() { //这里的source就是我们自定义的
observable对象，即observableCreate
source.subscribe(parent); } }
```

很简单的一个类，就是实现了 `Runnable` 接口，然后 `run()` 中调用 `Observer.subscribe()`。

### 5.2.1.7 Scheduler类的scheduleDirect()

再来看下 `scheduler.scheduleDirect()` 方法

```
public Disposable scheduleDirect(@NonNull Runnable
run) { return scheduleDirect(run, 0L,
TimeUnit.NANOSECONDS); }
```

往下看：

```

 public Disposable scheduleDirect(@NonNull Runnable
run, long delay, @NonNull TimeUnit unit) {
//createworker()在Scheduler类中是个抽象方法，所以其具体实现在其
子类中 //因此这里的createworker()应当是在IOScheduler中
实现的。 //worker中可以执行Runnable final
Worker w = createworker(); //实际上
decoratedRun还是这个run对象，即SubscribeTask final
Runnable decoratedRun = RxJavaPlugins.onSchedule(run);
 //将Runnable和worker包装成一个DisposeTask
DisposeTask task = new DisposeTask(decoratedRun, w);
 //worker执行这个task w.schedule(task,
delay, unit); return task; }

```

我们来看下创建worker和worker执行任务的过程。

### 5.2.1.8 IoScheduler的createWorker()和schedule()

```

 final AtomicReference<CachedworkerPool> pool;
public Worker createworker() { //就是new一个
EventLoopWorker, 并且传一个worker缓存池进去 return new
EventLoopWorker(pool.get()); } static final
class EventLoopWorker extends Scheduler.Worker {
private final CompositeDisposable tasks; private
final CachedworkerPool pool; private final
Threadworker threadworker; final AtomicBoolean
once = new AtomicBoolean(); //构造方法
EventLoopWorker(CachedworkerPool pool) {
this.pool = pool; this.tasks = new
CompositeDisposable(); //从缓存worker池中取一个
Worker出来 this.threadworker = pool.get();
} @NonNull @Override public
Disposable schedule(@NonNull Runnable action, long
delayTime, @NonNull TimeUnit unit) { //省略无关
代码 //Runnable交给threadworker去执行
 return threadworker.scheduleActual(action,
delayTime, unit, tasks); } }

```

注意，不同的scheduler类会有不同的worker实现，因为scheduler类最终是交到worker中去执行调度的。

我们来看下worker缓存池的操作：

### 5.2.1.9 CachedWorkerPool的get()

```
static final class CachedWorkerPool implements
Runnable { Threadworker get() { if
(allworkers.isDisposed()) { return
SHUTDOWN_THREAD_WORKER; } while
(!expiringworkerQueue.isEmpty()) { //如果缓
冲池不为空，就从缓存池中取threadworker
Threadworker threadworker = expiringworkerQueue.poll();
 if (threadworker != null) {
 return threadworker; } }
 //如果缓冲池中为空，就创建一个并返回。
 Threadworker w = new Threadworker(threadFactory);
 allworkers.add(w); return w; } }
```

### 5.2.1.10 NewThreadWorker的scheduleActual()

我们再来看下threadworker.scheduleActual()。

Threadworker类没有实现scheduleActual()方法，其父类NewThreadworker实现了该方法，我们点进去看下：

```

public class NewThreadworker extends Scheduler.Worker
implements Disposable { private final
ScheduledExecutorService executor; volatile boolean
disposed; public NewThreadworker(ThreadFactory
threadFactory) { //构造方法中创建一个
ScheduledExecutorService对象，可以通过
ScheduledExecutorService来使用线程池 executor =
SchedulerPoolFactory.create(threadFactory); }
public ScheduledRunnable scheduleActual(final Runnable
run, long delayTime, @NotNull TimeUnit unit, @Nullable
DisposableContainer parent) { //这里的decoratedRun实
际还是run对象 Runnable decoratedRun =
RxJavaPlugins.onSchedule(run); //将decoratedRun包装
成一个新对象ScheduledRunnable ScheduledRunnable sr =
new ScheduledRunnable(decoratedRun, parent); //省略
无关代码 if (delayTime <= 0) { //线程池中立即执行ScheduledRunnable f =
executor.submit((Callable<Object>)sr); } else {
 //线程池中延迟执行ScheduledRunnable f =
executor.schedule((Callable<Object>)sr, delayTime, unit);
 } //省略无关代码 return
sr; }
}

```

这里的 executor 就是使用线程池去执行任务，最终 subscribeTask 的 run() 方法会在线程池中被执行，即 observable 的 subscribe() 方法会在 IO 线程中被调用。这与上面例子中的输出结果符合：

```

Observable subscribe() 所在线程为 :RxCachedThreadScheduler-1

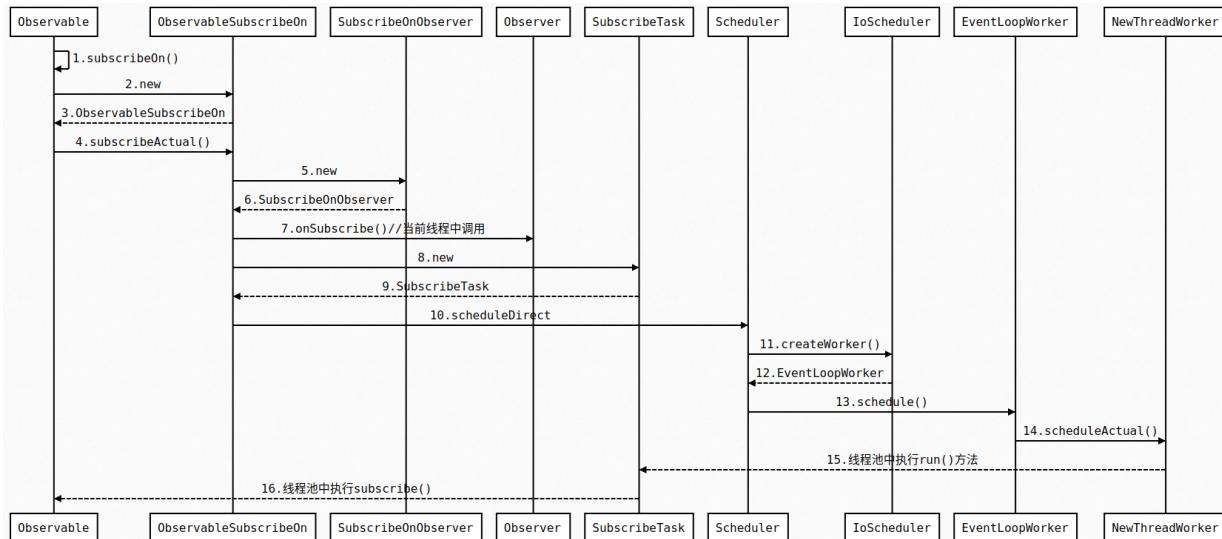
```

### 5.2.1.11 简单总结

1. `Observer` (观察者) 的 `onSubscribe()` 方法运行在当前线程中，因为在这之前都没涉及到线程切换。
2. 如果设置了 `subscribeOn(指定线程)`，那么 `observable` (被观察者) 中 `subscribe()` 方法将会运行在这个指定线程中去。

### 5.2.1.12 时序图

## 来张总的 subscribeOn() 切换线程时序图



### 5.2.1.13 多次设置 subscribeOn() 的问题

如果我们多次设置 subscribeOn()，那么其执行线程是在哪一个呢？先来看下例子

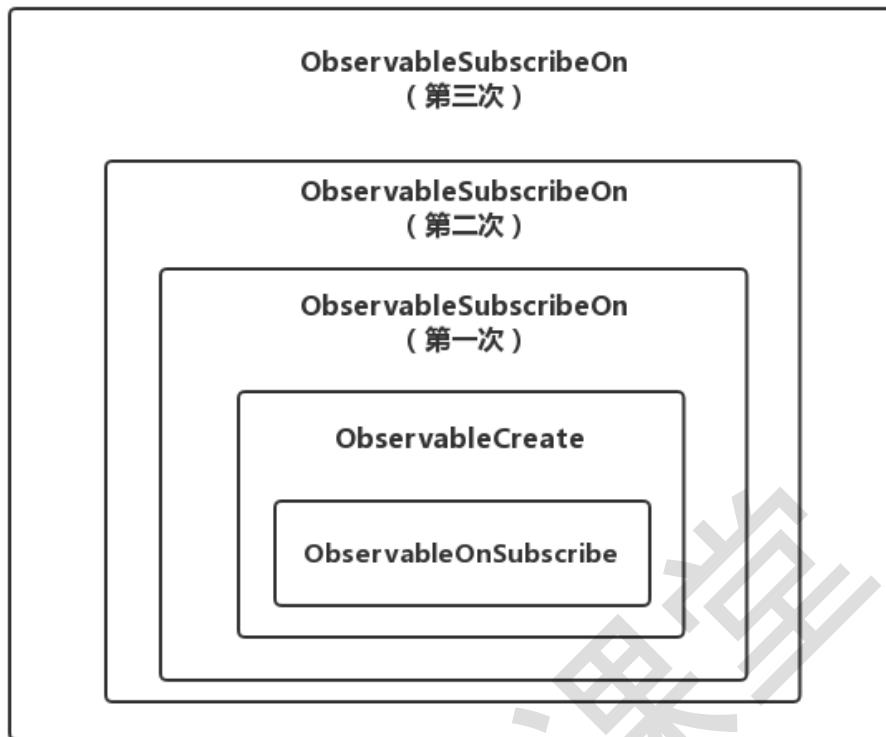
```
//省略前后代码，看重点部分
.subscribeOn(Schedulers.io())//第一次
.subscribeOn(Schedulers.newThread())//第二次
.subscribeOn(AndroidSchedulers.mainThread())//第三次
```

其输出结果为：

```
Observable subscribe() 所在线程为 :RxCachedThreadsScheduler-1
```

即只有第一次的 `subscribeOn()` 起作用了。这是为什么呢？

我们知道，每调用一次 `subscribeOn()` 就会把旧的被观察者包装成一个新的被观察者，经过了三次调用之后，就变成了下面这个样子：



同时，我们知道，被观察者被订阅时是从最外面的一层通知到里面的一层，那么当传到上图第三层时，也就是 `ObservableSubscribeOn` (第一次) 那一层时，管你之前是在哪个线程，

`subscribeOn(Schedulers.io())` 都会把线程切到 IO 线程中去执行，所以多次设置 `subscribeOn()` 时，只有第一次生效。

## 5.2.2 observeOn()

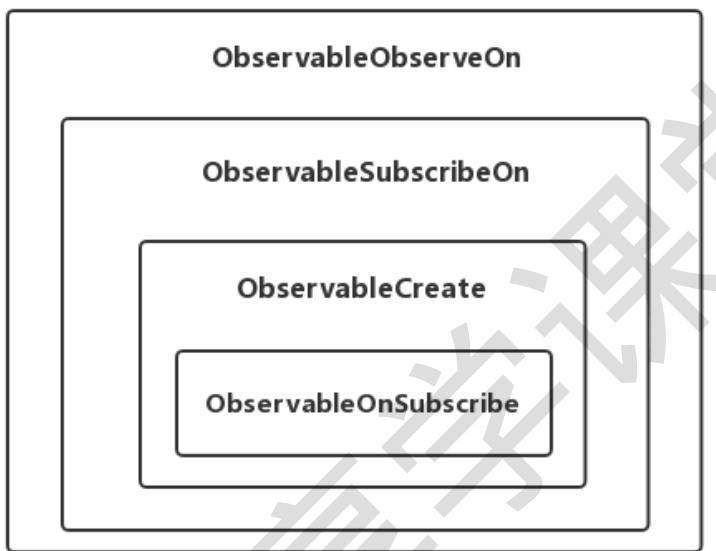
我们再来看下 `observeOn()`，还是先来回顾一下我们例子中的设置：

```
// 指定在Android主线程中执行
.observeOn(AndroidSchedulers.mainThread())
```

### 5.2.2.1 Observable类的observeOn()

```
public final Observable<T> observeOn(Scheduler scheduler) { return observeOn(scheduler, false, bufferSize()); } public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bufferSize) { //省略无关代码 return RxJavaPlugins.onAssembly(new ObservableobserveOn<T>(this, scheduler, delayError, bufferSize)); }
```

同样，这里也是新包装一个`observableobserveOn`对象，注意，这里包装的旧被观察者是`observableSubscribeOn`对象了，因为之前调用过`subscribeOn()`包装了一层了，所以现在是如下图所示：



`RxJavaPlugins.onAssembly()`也是原样返回。

我们看看`observableobserveOn`的构造方法。

### 5.2.2.2 ObservableOnSubscribeOn类的构造方法

```
public ObservableOnSubscribeOn(ObservableSource<T> source, Scheduler scheduler, boolean delayError, int bufferSize) { super(source); this.scheduler = scheduler; this.delayError = delayError; this.bufferSize = bufferSize; }
```

里面就是一些变量赋值而已。

### 5.2.2.3 ObservableOnSubscribeOn的subscribeActual()

和subscribeOn()差不多，我们就直接来看observableobserveOn的subscribeActual()方法了。

```
 @Override protected void
 subscribeActual(Observer<? super T> observer) { //判断是否当前线程 if (scheduler instanceof
 TrampolineScheduler) { //是当前线程的话，直接调用里
 面一层的subscribe()方法 //即调用
 observableSubscribeOn的subscribe()方法
 source.subscribe(observer); } else { //创建worker //本例子中的scheduler为
 AndroidSchedulers.mainThread()
 Scheduler.Worker w = scheduler.createWorker();
 //这里会将worker包装到ObserveOnObserver对象中去 //注意：source.subscribe没有涉及到worker，所以还是在之前设置的线程
 中去执行 //本例子中source.subscribe就是在IO线程中执
 行。 source.subscribe(new ObserveOnObserver<T>
 (observer, w, delayError, bufferSize)); } }
```

同样，这里也将observer给包装了一层，如下图所示：



source.subscribe()中将会把事件逐一发送出去，我们这里只看下observeonobserver中的onNext()方法的处理，onComplete()等就不看了，实际上都差不多。

#### 5.2.2.4 ObserveOnObserver的onNext()

```
 @Override public void onNext(T t) {
 //省略无关代码 if (sourceMode !=
 QueueDisposable.ASYNC) { //将信息存入队列中
 queue.offer(t); }
 schedule(); }
```

就是调用 `schedule()` 而已。

### 5.2.2.5 ObserveOnObserver的schedule()

```
 void schedule() { if
(getAndIncrement() == 0) {
 //ObserveOnObserver同样实现了Runnable接口，所以就把它自己交给
 worker去调度了 worker.schedule(this);
 } }
```

Android主线程调度器里面的代码就不分析了，里面实际上是用 `handler` 来发送 `Message` 去实现的，感兴趣的可以看下。

既然 `ObserveOnObserver` 实现了 `Runnable` 接口，那么就是其 `run()` 方法会在主线程中被调用。

我们来看下 `ObserveOnObserver` 的 `run()` 方法：

### 5.2.2.6 ObserveOnObserver的run()

```
 @Override public void run() {
 //outputFused默认是false if (outputFused) {
 drainFused(); } else {
 drainNormal(); } }
```

这里会走到 `drainNormal()` 方法。

### 5.2.2.7 ObserveOnObserver的drainNormal()

```
void drainNormal() { int missed = 1;
 //存储消息的队列 final SimpleQueue<T> q
 = queue; //这里的actual实际上是
 SubscribeOnObserver final Observer<? super T>
 a = actual; //省略无关代码
 //从队列中取出消息 v = q.poll();
 //...
 //这里调用的是里面一层的
 onNext()方法 //在本例子中，就是调用
 SubscribeOnObserver.onNext() a.onNext(v);
 //...
}
```

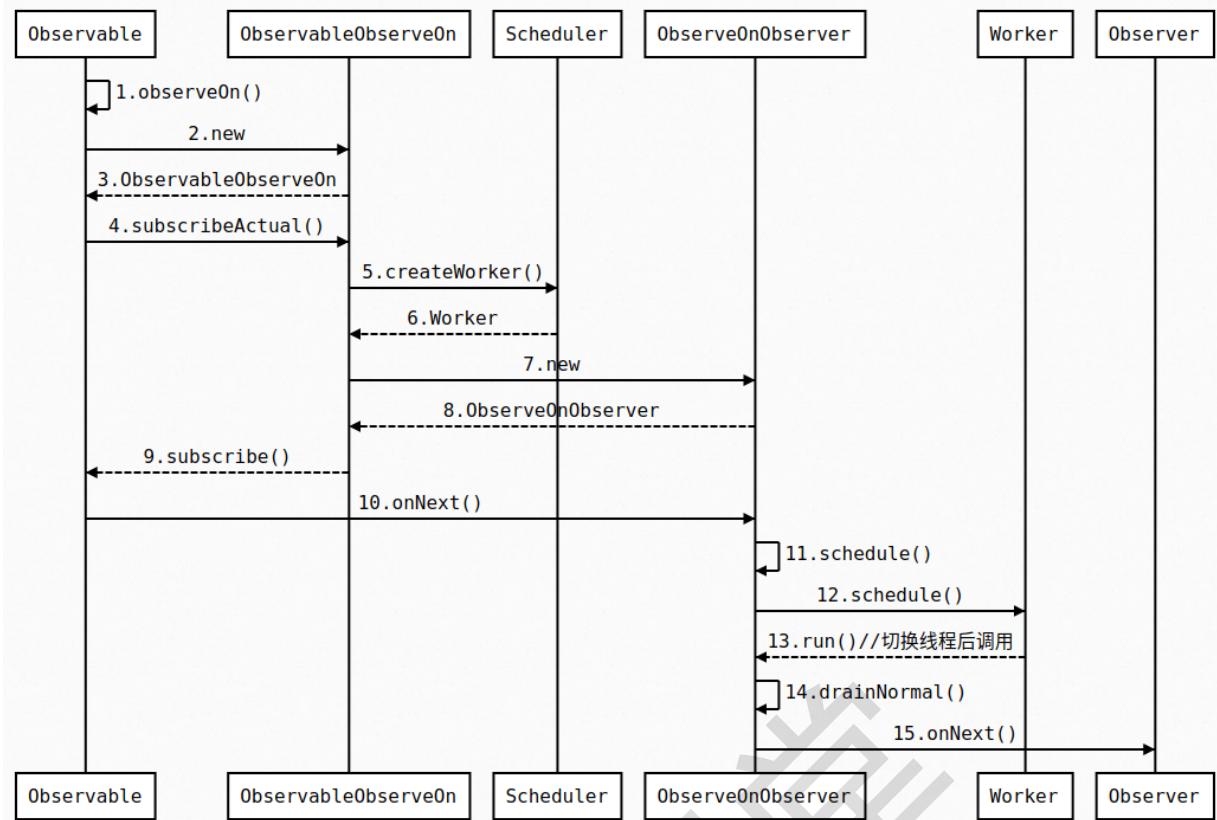
至于 `SubscribeOnObserver.onNext()`，里面也没切换线程的逻辑，就是调用里面一层的 `onNext()`，所以最终会调用到我们自定义的 `Observer` 中的 `onNext()` 方法。因此，`Observer` 的 `onNext()` 方法就在 `observeOn()` 中指定的线程中给调用了，在本例中，就是在Android主线程中给调用。

### 5.2.2.8 简单总结

1. 如果设置了 `observeOn(指定线程)`，那么 `Observer`（观察者）中的 `onNext()`、`onComplete()` 等方法将会运行在这个指定线程中去。
2. `subscribeOn()` 设置的线程不会影响到 `observeOn()`。

### 5.2.2.9 时序图

最后，来张observeOn()时序图：



## 7.20.2 Android：图文解析带你快速了解RxJava原理

### 1. 定义

- RxJava 在 GitHub 的介绍：

RxJava: a library for composing asynchronous and event-based programs using observable sequences for the Java VM// 翻译：RxJava 是一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库

- 总结：RxJava 是一个 **基于事件流、实现异步操作的库**

- o \*

### 2. 作用

实现异步操作

类似于 Android 中的 `AsyncTask`、`Handler` 作用

### 3. 特点

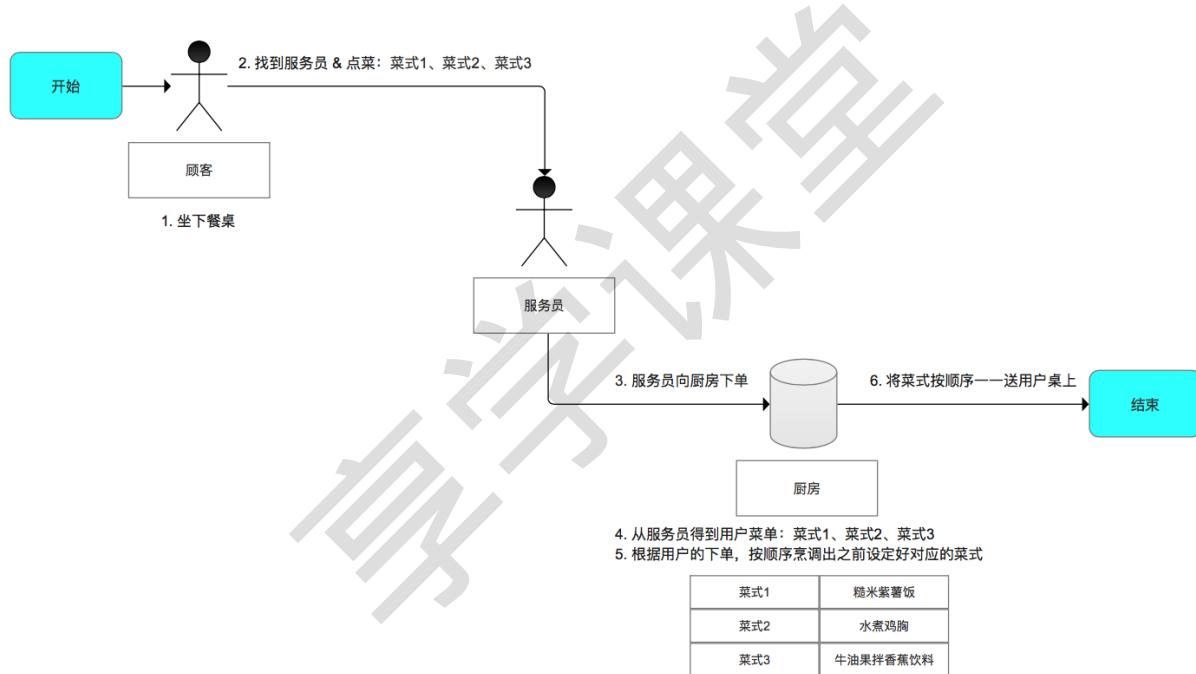
由于 RxJava 的使用方式是：**基于事件流的链式调用**，所以使得 RxJava：

- 逻辑简洁
- 实现优雅
- 使用简单

更重要的是，随着程序逻辑的复杂性提高，**它依然能够保持简洁 & 优雅**

### 4. 生活例子引入

我用一个生活例子引入 & 讲解 Rxjava 原理：顾客到饭店吃饭



示意图



示意图

### 5. 原理介绍

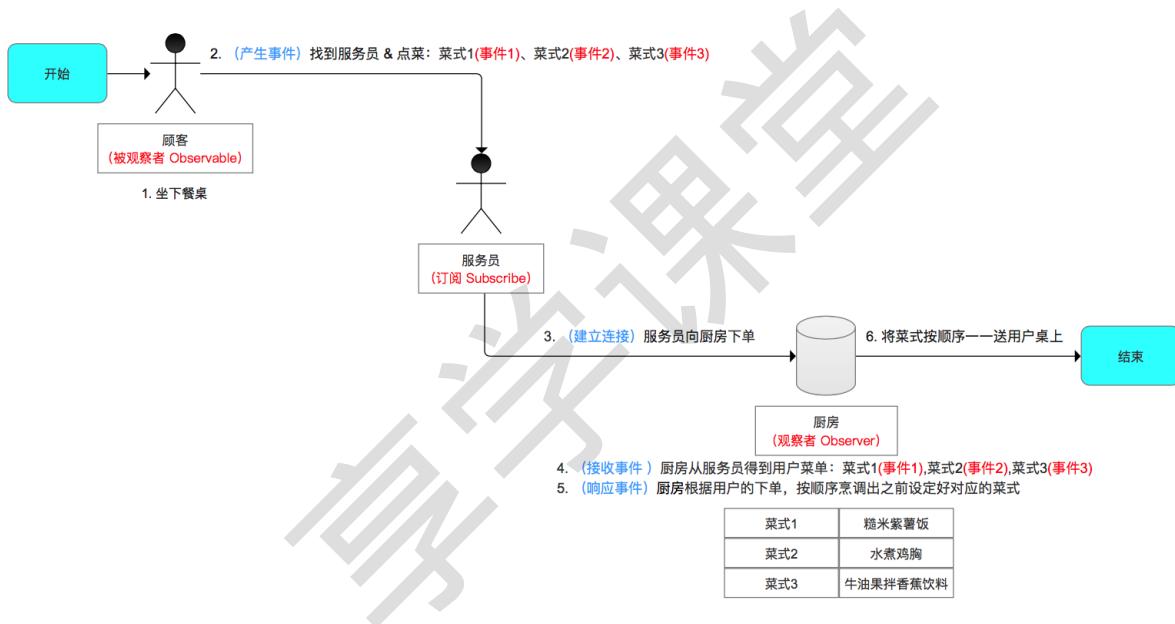
#### 5.1 概述

RxJava 原理 基于一种扩展的观察者模式，整个模式中有4个角色：

| 角色                | 作用               | 类比  |
|-------------------|------------------|-----|
| 被观察者 (Observable) | 产生事件             | 顾客  |
| 观察者 (Observer)    | 接收事件，并给出响应动作     | 厨房  |
| 订阅 (Subscribe)    | 连接 被观察者 & 观察者    | 服务员 |
| 事件 (Event)        | 被观察者 & 观察者 沟通的载体 | 菜式  |

## 5.2 具体描述

请结合上述 顾客到饭店吃饭 的生活例子理解：



示意图

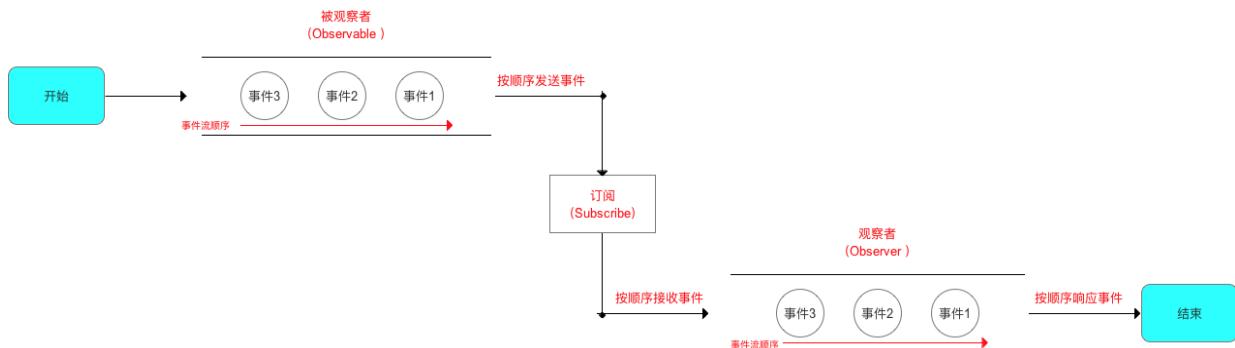


示意图

## 5.3 总结

RxJava 原理可总结为：

- 被观察者 (Observable) 通过订阅 (Subscribe) 按顺序发送事件给观察者 (Observer)
- 观察者 (Observer) 按顺序接收事件 & 作出对应的响应动作。具体如下图：



示意图

至此，关于 RxJava 原理讲解完毕。

## 6. 总结

- 本文主要对 RxJava 的原理进行图文讲解

## 20.2 RxJava如何进行线程切换的？

### 20.2.1 RxJava 是如何实现线程切换的（上）

#### 前言

通过前一篇的[从观察者模式出发，聊聊RxJava](#),我们大致理解了RxJava的实现原理，在RxJava中可以非常方便的实现不同线程间的切换。

`subscribeOn` 用于指定上游线程，`observeOn` 用于指定下游线程，多次用 `subscribeOn` 指定上游线程只有第一次有效，多次用 `observeOn` 指定下次线程，每次都有效；简直太方便了，比直接使用Handler省了不少力气，同时也不用去关注内存泄漏的问题了。本篇就来看看在RxJava中上游是如何实现线程切换。

#### RxJava 基础原理

为了方便后面的叙述，这里通过下面的UML图简单回顾一下上一篇的内容。

此图并没有完整的展现图中各个接口和类之间的各种关系，因为那样会导致整个图错综复杂，不便于查看，这里只绘制出了RxJava各个类之间核心关系网络

从上面的UML图中可以看出，具体的实现类只有ObservableCreate和CreateEmitter。CreateEmitter是ObservableCreate的内部类  
(PlantUML 怎么绘制内部类，没搞懂，玩的转的同学请赐教呀(▽))。

上篇说过Observable创建的过程，可以简化如下：

```
Observable mobservable=new ObservableCreate(new
ObservableOnSubscribe())
```

结合图可以更直观的体现出这一点。ObservableCreate 内部持有 ObservableOnSubscribe的引用。

当观察者订阅主题后：

```
mobservable.subscribe(mobserver);
```

ObservableCreate 中的subscribeActual()方法就会执行，

```
protected void subscribeActual(Observer<? super T>
observer) { CreateEmitter<T> parent = new
CreateEmitter<T>(observer);
observer.onSubscribe(parent); try {
source.subscribe(parent); } catch (Throwable ex) {
Exceptions.throwIfFatal(ex);
parent.onError(ex); } }
```

在这个过程中会创建CreateEmitter 的实例，而这个CreateEmitter实现了 Emitter和Disposable接口，同时又持有Observer的引用（当然这个引用是ObservableCreate传递给他的）。接着就会执行

**ObservableOnSubscribe的subscribe 方法**，方法的参数即为刚刚创建的CreateEmitter 的实例，接着一系列连锁反应，Emitter 接口中的方法 (onNext, onComplete等) 开始执行，在CreateEmitter内部，Observer

接口中对应的方法依次执行，这样就实现了一次从主题（上游）到观察者（下游）的事件传递。

### **source.subscribe(parent)**

这里的 source 是 ObservableOnSubscribe 的实例，parent 是 CreateEmitter 的实例。上面加粗文本叙述的内容，就是这行代码，可以说这是整个订阅过程最核心的实现。

好了，回顾完基础知识后，马上进入正题，看看 RxJava 是如何实现线程切换的。

### **RxJava 之 subscribeOn**

我们知道正常情况下，所有的内容都是在主线程执行，既然这里提到了线程切换，那么必然是切换到了子线程，因此，这里需要关注线程的问题，我们就带着下面这几个问题去阅读代码。

- 1. **是哪个对象在什么时候创建了子线程，是一种怎样的方式创建的？**
- 2. **子线程又是如何启动的？**
- 3. **上游事件是怎么跑到子线程里执行的？**
- 4. **多次用 subscribeOn 指定上游线程为什么只有第一次有效？**

### **示例**

首先看一下，日常开发中实现线程切换的具体实现

```
private void multiThread() {
 observable.create(new ObservableOnSubscribe<String>() {
 @Override public void
 subscribe(ObservableEmitter<String> e) throws Exception {
 e.onNext("This msg from work thread :" +
 Thread.currentThread().getName());
 sb.append("\nsubscribe: currentThreadName==" +
 Thread.currentThread().getName());
 }
 .subscribeOn(Schedulers.newThread())
 .observeOn(AndroidSchedulers.mainThread())
 .subscribe(new Consumer<String>() {
 @Override public void
 accept(String s) throws Exception {
 Log.e(TAG, "accept: s= " + s);
 }
 });
 }
}
```

这段代码，使用过RxJava的同学再熟悉不过了，上游事件会在一个名为 RxNewThreadScheduler-1 的线程执行，下游线程会切换回我们熟悉的 Android UI线程。

我们就从subscribeOn(Schedulers.newThread()) 出发，看看这个代码的背后，到底发生了什么。

## subscribeOn

这里我们先不管Schedulers.newThread() 是什么鬼，首先看看这个 subscribeOn()方法。

### Observable.java--- subscribeOn(Scheduler scheduler)

```
public final Observable<T> subscribeOn(Scheduler
scheduler) {
 ObjectHelper.requireNonNull(scheduler, "scheduler is
 null"); return RxJavaPlugins.onAssembly(new
 ObservableSubscribeOn<T>(this, scheduler));
}
```

可以看到，这个方法需要一个Scheduler 类型的参数。

### RxJavaPlugins.java--- onAssembly(@NonNull Observable source)

```
public static <T> Observable<T> onAssembly(@NotNull
Observable<T> source) { Function<? super
Observable, ? extends Observable> f =
onobservableAssembly; if (f != null) {
return apply(f, source); } return source;
}
```

O(∩\_∩)O哈哈~, 是不是觉得似曾相识, 和create操作符一个套路呀。因此, observeOn也可以简化如下:

```
new ObservableSubscribeOn<T>(this,
Schedulers.newThread());
```

这里你也许会有疑问, 这个this是什么呢? 其实这个this就是Observable, 具体到上面的代码来说就是ObservableCreate, 总之就是一个具体的Observable。

接着看ObservableSubscribeOn这个类

```
public final class ObservableSubscribeOn<T> extends
AbstractObservableWithUpstream<T, T> {}
```

看一下 **AbstractObservableWithUpstream.java**

```
abstract class AbstractObservableWithUpstream<T, U>
extends Observable<U> implements
HasUpstreamObservableSource<T> { /** The source
consumable observable. */ protected final
ObservableSource<T> source;
AbstractObservableWithUpstream(ObservableSource<T>
source) { this.source = source; } @Override
public final ObservableSource<T> source() {
return source; }}
```

再看一下 **HasUpstreamObservableSource.java**

```
/** * Interface indicating the implementor has an upstream ObservableSource-like source available * via {@link #source()} method. * * @param <T> the value type */public interface HasUpstreamObservableSource<T> {
 /** * Returns the upstream source of this Observable.
 * <p>Allows discovering the chain of observables.
 * @return the source ObservableSource */
 ObservableSource<T> source();}
```

饶了半天，`ObservableSubscribeOn` 原来和上一篇说的 `ObservableCreate`一样，也是 `Observable` 的一个子类。只不过比 `ObservableCreate` 多实现了一个接口 `HasUpstreamObservableSource`，这个接口很有意思，他的 `source()` 方法返回类型是 `ObservableSource`（还记得这个类的角色吗？）。也就是说 `ObservableSubscribeOn` 这个 `Observable` 是一个拥有上游的 `Observable`。他有一个非常关键的属性 `source`，这个 `source` 就代表了他的上游。

我们接着看 `ObservableSubscribeOn` 的具体实现。

```
public final class ObservableSubscribeOn<T> extends AbstractObservableWithUpstream<T, T> { final Scheduler scheduler;
 public ObservableSubscribeOn(ObservableSource<T> source,
 Scheduler scheduler) { super(source);
 this.scheduler = scheduler; } @Override public void subscribeActual(final Observer<? super T> s) {
 final SubscribeOnObserver<T> parent = new
 SubscribeOnObserver<T>(s); // observer 调用
 onSubscribe方法，获取上游的控制权
 s.onSubscribe(parent);
 parent.setDisposable(scheduler.scheduleDirect(new
 SubscribeTask(parent))); }}
```

- 首先看他的构造函数，参数 `source` 就是我们之前提到过的 `this`，`scheduler` 就是 `Schedulers.newThread()`。同时调用了父类 `AbstractObservableWithUpstream` 的构造函数，这里结合之前的结

论，我们可以确定通过这个构造函数，就创建出来了一个包含上游的 ObservableSubscribeOn实例。

- 再看实现订阅关系的关键方法subscribeActual()，在这里创建了一个 SubscribeOnObserver的实例，SubscribeOnObserver 是 AtomicReference的子类(保证原子性)，同时实现了 Observer接口 和 Disposable 接口；你可以把他理解成一个Observer。

我们之前说过，subscribeActual()是实现上下游之间订阅关系的重要方法。因为只有真正实现了订阅关系，上下游之间才能连接起来。我们看这个方法的最后一句代码。

```
parent.setDisposable(scheduler.scheduleDirect(new
SubscribeTask(parent)));
```

这句代码，可以说就是非常关键，因为从这里开始了一系列的连锁反应。首先看一下SubscribeTask

```
final class SubscribeTask implements Runnable {
private final SubscribeOnObserver<T> parent;
SubscribeTask(SubscribeOnObserver<T> parent) {
this.parent = parent; } @Override
public void run() { source.subscribe(parent);
} }
```

看到这句 **source.subscribe(parent)**，是不是觉得似曾相识呢？

SubscribeTask 实现了是Runnable接口，在其run方法中，定义了一个需要在线程中执行的任务。按照类的继承关系，很明显source 就是 ObservableSubscribeOn 的上游Observable，parent是一个 Observer。也就是说这个run方法要执行的内容就是实现 ObservableSubscribeOn的上游和Observer的订阅。**一旦某个线程执行了这个Runnable (SubscribeTask) ，就会触发了这个run方法，从而实现订阅**，而一旦这个订阅实现，那么后面的流程就是上节所说的事情了。

这里可以解答第三个问题了，上游事件是怎么给弄到子线程里去的，这里很明显了，就是直接把订阅方法放在了一个Runnable中去执行，这样就一旦这个Runnable在某个子线程执行，那么上游所有事件只能在这个子线程中执行了。

好了，线程要执行的任务似乎创建完了，下面就接着找看看子线程是怎么创建的。回过头继续看刚才的方法，

```
scheduler.scheduleDirect(new SubscribeTask(parent))
```

### Scheduler.java----scheduleDirect

```
public Disposable scheduleDirect(@NonNull Runnable run) { return scheduleDirect(run, 0L, TimeUnit.NANOSECONDS); } public Disposable scheduleDirect(@NonNull Runnable run, long delay, @NonNull TimeUnit unit) { final Worker w = createWorker(); // 对run进行了一次装饰 final Runnable decoratedRun = RxJavaPlugins.onSchedule(run); DisposeTask task = new DisposeTask(decoratedRun, w); w.schedule(task, delay, unit); return task; } @NonNull // 抽象方法 public abstract Worker createWorker();
```

首先看一下Worker类

```
/** * Sequential Scheduler for executing actions on a single thread or event loop. * <p> * Disposing the {@link Worker} cancels all outstanding work and allows resource cleanup. */ public abstract static class Worker implements Disposable { @NonNull public Disposable schedule(@NonNull Runnable run) { return schedule(run, 0L, TimeUnit.NANOSECONDS); } @NonNull public abstract Disposable schedule(@NonNull Runnable run, long delay, @NonNull TimeUnit unit); }
```

Worker是Scheduler内部的一个静态抽象类，实现了Disposable接口，其schedule()方法也是抽象的。

## 再看一下DisposeTask

```
static final class DisposeTask implements Runnable,
Disposable { final Runnable decoratedRun;
final Worker w; Thread runner;
DisposeTask(Runnable decoratedRun, Worker w) {
this.decoratedRun = decoratedRun; this.w = w;
} @Override public void run() {
runner = Thread.currentThread(); try {
decoratedRun.run(); } finally {
dispose(); runner = null;
} } @Override public void
dispose() { if (runner ==
Thread.currentThread() && w instanceof NewThreadWorker) {
((NewThreadWorker)w).shutdown();
} else { w.dispose(); }
} @Override public boolean isDisposed() {
return w.isDisposed(); } }
```

DisposeTask 又是一个Runnable，同时也实现了Disposable接口。可以看到在他的run方法中会执行decoratedRun的run方法，这个decoratedRun其实就是参数中传递进来的run，也就是说，执行了这个**DisposeTask的run方法，就会触发SubscribeTask中的run方法**，因此，我们就要关注是谁执行了这个DisposeTask。

## 回到scheduleDirect()方法

```
public Disposable scheduleDirect(@NotNull Runnable
run, long delay, @NotNull TimeUnit unit) { final
Worker w = createWorker(); // 对run进行了一次装饰
final Runnable decoratedRun =
RxJavaPlugins.onSchedule(run); DisposeTask task =
new DisposeTask(decoratedRun, w); w.schedule(task,
delay, unit); return task; }
```

scheduleDirect()方法的实现我们总结一下：

1. 创建一个Worker对象w,而在Scheduler类中createWorker()方法被定义为抽象方法，因此我们需要去Scheduler的具体实现中了解这个

Worker的具体实现。

2. 对参数run通过RxJavaPlugins进行一次装饰，生成一个decoratedRun的Runnable（通过源码可以发现，其实什么也没干，就是原样返回）
3. 通过decoratedRun和w生成一个DisposeTask对象task
4. **通过Worker的schedule方法开始执行这个task。**

$\varepsilon = (\text{'o'} ^ *)$ )唉，说了这么久，子线程是如何创建的依然不清楚，无论是SubscribeTask还是DisposeTask只是定义会在某个子线程中执行的任务，并不代表子线程已被创建。但是通过以上代码，我们也可以收获一些有价值的结论：

- 最终的Runnable任务，将由某个具体的Worker对象的scheduler()方法执行。
- 这个scheduleDirect会返回一个Disposable对象，这样我们就可以通过Observer去控制整个上游的执行了。

好了，到这里对于subscribeOn()方法的分析已经到了尽头，我们找了最终需要运行子任务的对象Worker，而这个Worker是个抽象类，因此我们需要关注Worker的具体实现了。

下面我们就从刚才丢下的Schedulers.newThread() 换个角度来分析，看看能不能找到这个Worker的具体实现。

### Schedulers.newThread()

前面说了subscribeOn()方法需要一个Scheduler类型的参数，然而通过前面的分析我们知道Scheduler是个抽象类，是无法被实例化的。因此，这里就从Schedulers类出发。

```
/** * static factory methods for returning standard
Scheduler instances. */public final class Schedulers {}
```

注释很清楚，这个Schedulers就是一个用于生成Scheduler实例的静态工厂。

下面我们就来看看，在这个工厂中newThread() 生成了一个什么样的Scheduler实例。

```
@NotNull public static Scheduler newThread() {
 return RxJavaPlugins.onNewThreadscheduler(NEW_THREAD);
}
 NEW_THREAD =
RxJavaPlugins.initNewThreadscheduler(new
NewThreadTask()); static final class NewThreadTask
implements Callable<Scheduler> { @Override
public Scheduler call() throws Exception {
 return NewThreadHolder.DEFAULT; } } static
final class NewThreadHolder { static final
Scheduler DEFAULT = new NewThreadscheduler(); }
```

newThread() 方法经过层层委托处理(最终的创建方式，有点单例模式的意味)，最终我们需要的就是一个NewThreadScheduler的实例。

## NewThreadScheduler.java

```
public final class NewThreadscheduler extends Scheduler {
 final ThreadFactory threadFactory; private static
final String THREAD_NAME_PREFIX = "RxNewThreadscheduler";
 private static final RxThreadFactory THREAD_FACTORY;
/** The name of the system property for setting the
thread priority for this scheduler. */ private static
final String KEY_NEWTTHREAD_PRIORITY = "rx2.newthread-
priority"; static { int priority =
Math.max(Thread.MIN_PRIORITY,
Math.min(Thread.MAX_PRIORITY,
Integer.getInteger(KEY_NEWTTHREAD_PRIORITY,
Thread.NORM_PRIORITY))); THREAD_FACTORY = new
RxThreadFactory(THREAD_NAME_PREFIX, priority); }
public NewThreadscheduler() {
this(THREAD_FACTORY); } public
NewThreadscheduler(ThreadFactory threadFactory) {
this.threadFactory = threadFactory; } @NotNull
@Override public Worker createWorker() { return
new NewThreadWorker(threadFactory); }}
```

不出所料NewThreadScheduler 是Scheduler的一个子类，在他的静态代码块中构造了一个Priority=5的线程工厂。而在我们最最关注的**createWorker()**方法中他又用这个线程工厂创建了一个**NewThreadWorker** 的实例。下面就让我们看看最终的NewThreadWorker 做了些什么工作。

### NewThreadWorker.java(节选关键内容)

```
public class NewThreadworker extends Scheduler.Worker
implements Disposable { private final
ScheduledExecutorService executor; volatile boolean
disposed; public NewThreadworker(ThreadFactory
threadFactory) { executor =
SchedulerPoolFactory.create(threadFactory); }
@NotNull @Override public Disposable
schedule(@NotNull final Runnable run) { return
schedule(run, 0, null); } @Override public
void dispose() { if (!disposed) {
disposed = true;
executor.shutdownNow();
} }}
```

众里寻他千百度，终于找到了Worker的实现了，同时再一次不出所料的又一次实现了Disposable接口，o(╥﹏╥)o。

在其构造函数中，通过NewThreadScheduler中提供的线程工厂threadFactory创建了一个ScheduledExecutorService。

### ScheduledExecutorService.java ---create

```
public static ScheduledExecutorService
create(ThreadFactory factory) { final
ScheduledExecutorService exec =
Executors.newScheduledThreadPool(1, factory);
if (PURGE_ENABLED && exec instanceof
ScheduledThreadPoolExecutor) {
ScheduledThreadPoolExecutor e =
(ScheduledThreadPoolExecutor) exec;
POOLS.put(e, exec); } return exec;
}
```

用大名鼎鼎的Executors(Executor的工具类)，创建了一个核心线程为1的线程。

**至此，我们终于找到了第一个问题的答案，子线程是谁如何创建的；在NewThreadScheduler的createWorker()方法中，通过其构建好的线程工厂，在Worker实现类的构造函数中创建了一个ScheduledExecutorService的实例，是通过SchedulerPoolFactory创建的。**

同时可以看到，通过执行dispose方法，可以使用ScheduledExecutorService的shutdown()方法，停止线程的执行。

线程已经创建好了，下面就来看看到底是谁启动了这个线程。前面我们说过，Worker的schedule()方法如果执行了，就会执行我们定义好的Runnable，通过这个Runnable中run方法的执行，就可以实现上下游订阅关系。下面就来看看这个scheduler()方法。

```
@NonNull @Override public Disposable
schedule(@NonNull final Runnable action, long delayTime,
@NonNull TimeUnit unit) { if (disposed) {
 return EmptyDisposable.INSTANCE; }
return scheduleActual(action, delayTime, unit, null);
} @NonNull public ScheduledRunnable
scheduleActual(final Runnable run, long delayTime,
@NonNull TimeUnit unit, @Nullable DisposableContainer
parent) { Runnable decoratedRun =
RxJavaPlugins.onSchedule(run); ScheduledRunnable
sr = new ScheduledRunnable(decoratedRun, parent);
if (parent != null) { if (!parent.add(sr)) {
 return sr; } }
Future<?> f; try { if (delayTime <= 0)
{ f =
executor.submit((Callable<Object>)sr); } else
{ f =
executor.schedule((Callable<Object>)sr, delayTime, unit);
 } sr.setFuture(f); } catch
(RejectedExecutionException ex) { if (parent
!= null) { parent.remove(sr); }
RxJavaPlugins.onError(ex); }
return sr; }
```

到这里，已经很明显了，在schedulerActual方法中，会通过刚才创建好的子线程对象executor通过submit或schedule执行一个Runnable任务（虽然这个Runnable对象再一次经过了各种装饰和包装，但其本质没有发生变化），并将执行结果封装后返回。而这个Runnable对象追根溯源来说，就是我们在ObservableSubscribeOn类中创建的一个SubscribeTask对象。因此，当这个子线程开始运行的时候就是执行SubscribeTask中run()方法的时机；一旦这个run方法执行，那么

```
source.subscribe(parent)
```

这句最关键的代码就开始执行了，一切的一切又回到了我们上一篇那熟悉的流程了。

好了，按照上面的流程捋下来，感觉还是有点分散，那么就用UML图看看整体的结构。

我们看最下面的ObservableSubscribeOn，他是subscribeOn 返回的 Observable对象，他持有一个Scheduler 实例的引用，而这个 Scheduler实例就是NewThreadScheduler(即 Schedulers.newThread())的一个实例。 ObservableSubscribeOn 的 subscribeActual方法，会触发NewThreadScheduler去执行 SubscribeTask中定义的任务，而这个具体的任务又将由Worker类创建的子线程去执行。这样就把上游事件放到了一个子线程中实现。

至于最后一个问題，多次用 subscribeOn 指定上游线程为什么只有第一次有效？，看完通篇其实也很好理解了，因为上游Observable只有一个任务，就是subscribe(准确的来说是subscribeActual())，而subscribeOn 要做的事情就是把上游任务切换到一个指定线程里，那么一旦被切换到了某个指定的线程里，后面的切换不就是没有意义了吗。

好了，至此上游事件切换到子线程的过程我们就明白了。下游事件又是如何切换的且听下回分解，本来想一篇写完的，结果发现越写越多，只能分成两篇了！！！ o(‘ □ ‘)o。

## 写在后面的话

### 关于Disposable

在RxJava的分析中，我们经常会遇到Disposable这个单词，确切的说是接口，这里简单说一说这个接口。

```
/** * Represents a disposable resource. */public
interface Disposable { void dispose(); boolean
isDisposed();}
```

我们知道，在Java中，类实现某个接口，通俗来说就是代表这个类多了一项功能，比如一个类实现Serializable接口，代表这个类是可以序列化的。这里Disposable也是代表一种能力，这个能力就是Disposable，就是代表一次性的，用后就丢弃的，比如一次性筷子，还有那啥。

在RxJava中很多类都实现了这个接口，这个接口有两个方法，`isDisposed()`顾名思义返回当前类是否被抛弃，`dispose()`就是主动抛弃。因此，所有实现了这个接口的类，都拥有了这样一种能力，就是可以判断自己是否被抛弃，同时也可以主动抛弃自己。

上一篇我们说了，`Observer`通过`onSubscribe(@NonNull Disposable d)`，会获得一个`Disposable`，这样就有能力控制上游的事件发送了。这样，我们就不难理解，为什么那么多类实现了这个接口，因为下游获取到的是一个拥有`Disposable`的对象，而一旦拥有了一个这样的对象，那么就可以通过下游控制上游了。可以说，这是RxJava对常规的观察者模式所做的最给力的改变。

## 关于各种ObservableXXX ,subscribeXXX,ObserverXXX

在查看RxJava的源码时，可能很多人都和我一样，有一个巨大的困扰，就是这些类的名字好他妈难记，感觉长得都差不多，关键念起来好像也差不多。但其实本质上来说，RxJava对类的命名还是非常规范的，只是我们不太习惯而已。按照英文单词翻译：

- `Observable` 可观察的
- `Observer` 观察者
- `Subscribe` 订阅

其实就这么三个主语，其他的什么`ObservableCreate`，`ObservableSubscribeOn`，`AbstractObservableWithUpstream`，还有上面提到的`Disposable`，都是对各种各样的`Observable`和`Observer`的变形和修饰结果，只要理解这个类的核心含义是什么，就不会被这些名字搞晕了。

RxJava 可以说是博大精深，以上所有分析完全是个人平时使用时的总结与感悟，有任何错误之处，还望各位读者提出，共同进步。

关于RxJava 这里墙裂推荐一篇文章[一篇不太一样的RxJava介绍](#)，感觉是自扔物线那篇之后，对RxJava思想感悟最深的一篇了。对RxJava 有兴趣的同学，可以多度几遍，每次都会有收获！！

## 20.2.2 RxJava 线程切换原理

### 基础概念

#### 观察者模式

##### 1. 观察者概念

观察者模式 (Observer Mode) 是定义对象间的一对多的依赖关系，当被观察者的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。

在观察者模式中有以下四个主要角色：

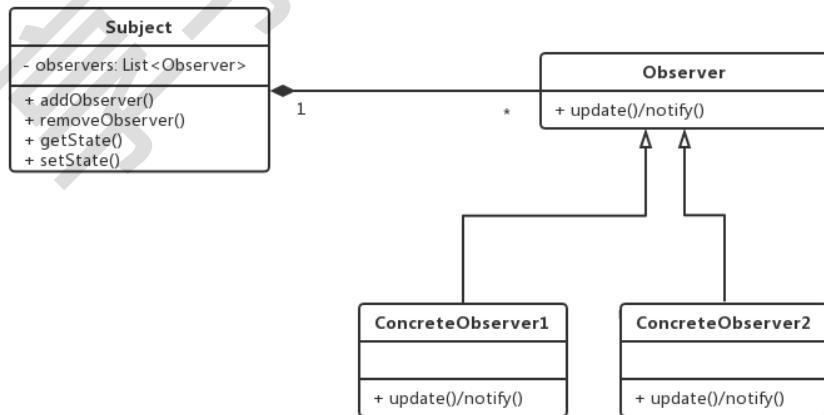
抽象主题[抽象被观察者] (Subject)：定义添加和删除观察者的方法，内部通过集合维护观察者序列。

具体主题[具体被观察者] (Concrete Subject)：抽象主题的实现对象，在具体主题内部状态发生变化时，通知所有的观察者更新状态。

抽象观察者 (Observer)：定义观察者的统一接口和方法。

具体观察者 (Concrete Observer)：抽象观察者的具体实现类，实现抽象观察者定义的统一接口，以便使本身的状态与主题状态协调。

经典的观察者模式UML类图：



##### 2. 观察者模式在Rxjava中的运用

为了方便分析问题，下面给出Rxjava实现的最简单的被观察者（主题）发送数据观察者打印数据的代码。从代码中分析Rxjava中是如何定义并且实现观察者模式中不同的角色。为了方便说明问题，把Rxjava中的链式(Chain)拆分成最基本的3段。

(1) Observable对象创建，抽象类Observable是接口

ObservableSource下的一个抽象实现，通过Observable创建一个可观察对象发射事件流。

(2) Observer对象创建，创建一个观察者Observer来接受并响应可观察对象发射的事件。

(3) Observer订阅Observable，通过subscribe方法，使Observer与Observable建立订阅关系，Observer与Observable便成为了一个整体，Observer便可对Observable中的行为作出响应。

PS：虽然从代码上看上去像是Observable订阅了Observer，但是其实还是观察者订阅了被观察者，Rxjava这么设计是为了保持链式调用(Chain)。

**这里问了说明问题，没有采用极简的代码实现。**

java实现：

```
private void rxjavaDemo() { Observable observable
= Observable.create(new ObservableOnSubscribe() {
 @Override public void
subscribe(ObservableEmitter e) throws Exception {
 e.onNext("R"); e.onNext("X");
 e.onComplete(); });
 observer observer = new observer() { @Override
 public void onSubscribe(Disposable d) {
 } @Override public void
onNext(Object s) {
Log.e(RxjavaDemoActivity.class.getSimpleName(), "object :
" + s); } @Override
 public void onError(Throwable e) { }
 @Override public void onComplete() {
 } observable.subscribe(observer); }
}
```

Kotlin实现：

```
private fun rxjavaDemo() {
 observable.create<String>{
 it.onNext("X")
 it.onComplete()
 }
 val mobserver = object : Observer<String>{
 override fun onComplete() {}
 override fun onSubscribe(d: Disposable?) {}
 override fun onNext(value: String?) {
 Log.e(RxjavaDemoActivity::class.java.simpleName,
"object : $value")
 }
 override fun onError(e: Throwable?) {}
 }
 mobservable.subscribe(mobserver)
}
```

### 3. 被观察者 (Observable)

在上面代码中我们调用了Observable的create方法来创建被观察者。

- (1) 在Observable类内部提供了众多的静态方法来创建被观察者。诸如: create、just、interval、from、zip、contact、merge等方法。
- (2) requireNonNull方法，是Rxjava的判空实现，防止出现空指针异常。
- (3) 在create方法中会创建ObservableCreate的被观察者。

```
@SchedulerSupport(SchedulerSupport.NONE) public
static <T> Observable<T> create(ObservableOnSubscribe<T>
source) { ObjectHelper.requireNonNull(source,
"source is null"); return
RxJavaPlugins.onAssembly(new ObservableCreate<T>
(source)); }
```

被观察者ObservableCreate类继承自抽象类Observable，内部实现了父类的subscribeActual方法。

```
public final class ObservableCreate<T> extends Observable<T>{..... @Override protected void subscribeActual(Observer<? super T> observer) { CreateEmitter<T> parent = new CreateEmitter<T>(observer); observer.onSubscribe(parent); try { source.subscribe(parent); } catch (Throwable ex) { Exceptions.throwIfFatal(ex); parent.onError(ex); } }
```

被观察者的抽象类Observable， Observable又是接口ObservableSource下的一个抽象实现。

- (1) 内部实现了ObservableSource接口定义的subscribe方法。  
subscribe方法内部主要是调用了subscribeActual方法，所有断定订阅关系是在subscribeActual方法内部实现的。
- (2) 所以要实现订阅关系，观察者真正需要复写的是subscribeActual方法。比如ObservableCreate类就复写了该方法。

```
public abstract class Observable<T> implements ObservableSource<T> {
@SchedulerSupport(SchedulerSupport.NONE) @Override
public final void subscribe(Observer<? super T> observer)
{ ObjectHelper.requireNonNull(observer, "observer
is null"); try { observer =
RxJavaPlugins.onSubscribe(this, observer);
ObjectHelper.requireNonNull(observer, "Plugin returned
null observer"); subscribeActual(observer);
} catch (NullPointerException e) { // NOPMD
throw e; } catch (Throwable e) {
Exceptions.throwIfFatal(e); // can't call
onError because no way to know if a Disposable has been
set or not // can't call onSubscribe because
the call might have set a Subscription already
RxJavaPlugins.onError(e); NullPointerException
npe = new NullPointerException("Actually not, but can't
throw other exceptions due to RS");
npe.initCause(e); throw npe; }
}.....}
```

Observable实现了ObservableSource接口，在ObservableSource内部定义了subscribe方法用来实现订阅观察者（Observer）。

```
public interface ObservableSource<T> { /** * Subscribes the given Observer to this observableSource * instance. * @param observer the Observer, not null * @throws NullPointerException if {@code observer} is null */ void subscribe(Observer<? super T> observer);}
```

总结：

- (1) ObservableSource就是扮演着抽象被观察者的角色。
- (2) 在ObservableSource 接口中定义了subscribe方法用来实现订阅观察者（Observer）。
- (3) Observable类实现了ObservableSource接口并且实现了其subscribe方法，但是它并没有真正的去完成主题和观察者之间的订阅关系，而是内部调用了另一个抽象方法subscribeActual。
- (4) 在Observable内部提供了一系列创建型操作符，用来创建不同场景的Observable。

经过上面的介绍，我们已经明白了在Rxjava中被观察者（Observable）是如何创建的，以及是谁扮演者抽象观察者的角色。但是我们并没有在ObservableCreate类中发现具体发送事件的实现。那么这里就有一个问题：

**问题： ObservableCreate等内部是如何发送事件到观察者（Observer）的？**

#### 4. 观察者（Observer）

通过上面的代码和分析，我们知道Observer扮演着抽象观察者的角色。

下面分别解释一下Observer类内部定义的四个主要的方法：

- (1) onSubscribe (Disposable d) 里面的Disposable对象，Disposable翻译过来是可随意使用的。相当于观察者和被观察者之间的订阅关系，如果观察者不想订阅被观察者了，可以调用mDisposable.dispose()取消订阅关系。
- (2) onCompleted(): 事件队列完成。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的onNext()

发出时，需要触发 onCompleted() 方法作为标志。

(3) onError(): 事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。

(4) onNext(): 接收数据。

(5) 在一个正确运行的事件序列中，onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。而且onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。

```
public interface Observer<T> { /** * Provides the
 * Observer with the means of cancelling (disposing) the
 * connection (channel) with the Observable in both *
 * synchronous (from within {@link #onNext(Object)}) and
 * asynchronous manner. * @param d the Disposable
 * instance whose {@link Disposable#dispose()} can * be
 * called anytime to cancel the connection * @since 2.0
 */
 void onSubscribe(Disposable d); /**
 * Provides the Observer with a new item to observe. *
 <p> * The {@link Observable} may call this method 0
 * or more times. * <p> * The {@code Observable}
 * will not call this method again after it calls either
 * {@link #onComplete} or * {@link #onError}.
 * * @param value * the item emitted by the
 * Observable */
 void onNext(T value); /**
 * Notifies the Observer that the {@link Observable} has
 * experienced an error condition. * <p> * If the
 * {@link Observable} calls this method, it will not
 * thereafter call {@link #onNext} or * {@link
 * #onComplete}. * * @param e * the
 * exception encountered by the Observable */
 void
 onError(Throwable e); /**
 * Notifies the Observer
 * that the {@link Observable} has finished sending push-
 * based notifications. * <p> * The {@link
 * Observable} will not call this method if it calls {@link
 * #onError}.
 */
 void onCompleted();}
```

既然Observer是个接口，那么就应该是个抽象观察者，具体的观察者是我们在实际运用的时候直接new一个实例对象。

经过上面的介绍，我们已经明白了在Rxjava中观察者是如何创建的以及各个方法的作用。那么观察者是如何接受被观察者发送的事件的呢？

**问题： Observer是如何接受数据到被观察者发送的数据？**

## 订阅

**前提：观察者订阅被观察者后，被观察者才会开始发送事件。**

PS：但是并不是所有的被观察者都需要被订阅才会发送数据，比如 Observable.just的方法返回的ObservableJust被观察者者。

示例：执行下面的代码会输出**JustObservable**，但是将just换成create方法就不会输出，所有得出结论：**并不是所有的被观察者都需要被订阅才会发送数据**

```
observable.just(new JustObservable());class
JustObservable{ JustObservable(){
 Log.e("rx","JustObservable"); }}
```

下面我们来分析上面遗留的两个问题：即**观察者和被观察者是如何发送和接受事件的**

Observable.create生成的被观察者需要被订阅后才会发送数据到观察者。

根据上面的分析我们知道：这里的observable对象是**ObservableCreate**类的实例。

```
observable.subscribe(observer);
```

这里的subscribe是父类Observable的方法，在里面又会调用 subscribeActual方法， Observable的子类ObservableCreate会复写 subscribeActual方法。

```

public final void subscribe(Observer<? super T> observer)
{ ObjectHelper.requireNonNull(observer, "observer
is null"); try { observer =
RxJavaPlugins.onSubscribe(this, observer);
ObjectHelper.requireNonNull(observer, "Plugin returned
null observer"); subscribeActual(observer);
} catch (NullPointerException e) { // NOPMD
throw e; } catch (Throwable e) {
Exceptions.throwIfFatal(e); // can't call
onError because no way to know if a Disposable has been
set or not // can't call onSubscribe because
the call might have set a Subscription already
RxJavaPlugins.onError(e); NullPointerException
npe = new NullPointerException("Actually not, but can't
throw other exceptions due to RS");
npe.initCause(e); throw npe; } } }

```

下面来分析一下ObservableCreate类。

- (1) 在ObservableCreate的构造方法中有个ObservableOnSubscribe类型的形参。
- (2) 并且正如我们上面所说内部实现了subscribeActual方法。
- (3) 所以真正处理被观察者和观察者之间实现订阅的逻辑在Observable的subscribeActual方法中。

```

public final class ObservableCreate<T> extends
Observable<T> { final ObservableOnSubscribe<T> source;
public ObservableCreate(ObservableOnSubscribe<T>
source) { this.source = source; } @Override
protected void subscribeActual(Observer<? super T>
observer) { CreateEmitter<T> parent = new
CreateEmitter<T>(observer);
observer.onSubscribe(parent); try {
source.subscribe(parent); } catch (Throwable ex) {
Exceptions.throwIfFatal(ex);
parent.onError(ex); } } }

```

那么在ObservableCreate的构造方法的形参的赋值肯定是在 ObservableCreate对象初始化的时候，然而ObservableCreate的初始化，是通过Observable的create方法，下面我们回到Observable的 create的方法。

```
public static <T> Observable<T>
create(ObservableOnSubscribe<T> source) {
 ObjectHelper.requireNonNull(source, "source is null");
 return RxJavaPlugins.onAssembly(new
 ObservableCreate<T>(source));
}
```

在调用create方法的时候需要传递ObservableOnSubscribe的对象作为参数，而这个对象最终会传入到ObservableCreate的构造方法中。

下面来看一下ObservableOnSubscribe的作用：

(1) 内部声明了一个subscribe方法，subscribe接收到一个 ObservableEmitter对象。

```
public interface ObservableOnSubscribe<T> { /**
 * called for each observer that subscribes. * @param e
 * the safe emitter instance, never null * @throws
 * Exception on error */ void
subscribe(ObservableEmitter<T> e) throws Exception;}
```

下面再来看一下ObservableEmitter类的作用：

(1) ObservableEmitter以一种可以安全取消的形式发送事件到观察者，通过调用setDisposable方法。

(2) 继承了Emitter接口。

```
public interface ObservableEmitter<T> extends Emitter<T>
{
 /** * Sets a Disposable on this emitter; any
 previous Disposable * or cancellation will be
 unsubscribed/cancelled. * @param d the disposable,
 null is allowed */ void setDisposable(Disposable
d); /** * Sets a Cancellable on this emitter; any
 previous Disposable * or cancellation will be
 unsubscribed/cancelled. * @param c the cancellable
 resource, null is allowed */ void
setCancellable(Cancellable c); /** * Returns true
 if the downstream disposed the sequence. * @return
 true if the downstream disposed the sequence */
 boolean isDisposed(); /** * Ensures that calls to
 onNext, onError and onComplete are properly serialized.
 * @return the serialized ObservableEmitter */
 ObservableEmitter<T> serialize();
}
```

下面我们来看一下Emitter类的作用：

(1) Emitter翻译过来就是发射器的意识，到这里我们可以想到，该类内部应该定义了一些跟事件发送相关的方法。

(2) 并且在实例化ObservableOnSubscribe的时候，我们正好是调用了ObservableEmitter的onNext向观察者发送数据的。

```
Observable observable = Observable.create(new
ObservableOnSubscribe() { @Override
public void subscribe(ObservableEmitter e) throws
Exception { e.onNext("R");
e.onNext("X"); e.onComplete();
} });

```

```
public interface Emitter<T> { /** * Signal a
normal value. * @param value the value to signal, not
null */ void onNext(T value); /** * Signal
a Throwable exception. * @param error the Throwable
to signal, not null */ void onError(Throwable
error); /** * Signal a completion. */ void
onComplete();}

```

由此我们可以知道Emitter内部声明了三种事件类型，而 ObservableEmitter 扩展了Emiiter的功能，添加了Disposable相关的方法，可以用来安全取消事件的发送即取消观察者和被观察者之间的订阅关系。

由上诉分析我们已经知道了ObservableCreate的 ObservableOnSubscribe变量的来历和基本作用以及被观察者的创建过程。

下面继续回到ObservableCreate类的subscribeActual方法来看看事件是如何从被观察者发送到观察者的。

(1) 在subscribeActual方法内部创建了CreateEmitter类对象，并且接受Observer作为参数，CreateEmitter实现了ObservableEmitter接口。所以该类是负责事件发送，到这里我们已经明确了事件的发送类即 ObservableEmitter。

(2) 在subscribeActual方法内部，调用了ObservableOnSubscribe的 subscribe方法并且传递ObservableEmitter对象实例作为参数，然后就可以调用ObservableEmitter的方法发送事件了。

(3) 在subscribeActual方法内部，调用了Observer的onSubscribe方法并且传递CreateEmitter作为参数，这样观察者就持有了发送事件（被观察者）的直接引用，方便观察者取消订阅关系。

到这里我们已经确定到了：观察者是如何和被观察者订阅的。以及事件是如何发送到被观察者的。并且确认了只有发生了订阅关系，事件才可以发送。

下面在来看一下CreateEmitter类的实现逻辑。

(1) CreateEmitter的构造函数，传递一个Observer的对象作为形参。这样就可以将事件发送到对应的观察者了。

(2) CreateEmitter实现了ObservableEmitter接口，作为事件发送器。

(3) onNext事件中，不可以发送参数为null的类型，在事件序列没有中断的情况下把事件从被观察者传递给观察者。

(4) onComplete事件，用于通知观察者事件队列已经没有事件发送了。

(5) onError事件，事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。

(6) setDisposable、setCancellable方法，观察者根据获取到的Emitter的实例对象，可以取消被观察者和观察者之间的订阅关系。

```
static final class CreateEmitter<T> extends
AtomicReference<Disposable> implements
ObservableEmitter<T>, Disposable { private static
final long serialVersionUID = -3434801548987643227L;
 final Observer<? super T> observer;
CreateEmitter(Observer<? super T> observer) {
this.observer = observer; } @Override
public void onNext(T t) { if (t == null) {
 onError(new NullPointerException("onNext
called with null. Null values are generally not allowed
in 2.x operators and sources.")); return;
 } if (!isDisposed()) {
 observer.onNext(t); } }
@Override public void onError(Throwable t) {
 if (t == null) { t = new
NullPointerException("onError called with null. Null
values are generally not allowed in 2.x operators and
sources."); } if (!isDisposed()) {
 try {
 observer.onError(t); } finally {
 dispose(); } } else {
 RxJavaPlugins.onError(t); } }
@Override public void onComplete() {
 if (!isDisposed()) { try {
 observer.onComplete(); } finally {
 dispose(); } } }
@Override public void
setDisposable(Disposable d) {
DisposableHelper.set(this, d); } @Override
public void setCancellable(Cancellable c) {
 setDisposable(new CancellableDisposable(c));
}
@Override public ObservableEmitter<T>
serialize() { return new SerializedEmitter<T>
(this); } @Override public void
dispose() { DisposableHelper.dispose(this);
}
@Override public boolean
isDisposed() { return
DisposableHelper.isDisposed(get()); } }
```

## 小结：

- (1) 在RxJava中Observer通过onSubscribe方法获取了发送事件中的Disposable对象，这样他就可以控制观察者和被观察者之间的订阅关系。
- (2) 被观察者并没有直接控制事件的发送，而是将事件的发送给Disposable对象的发送。
- (3) 订阅关系并没有发生在subscribe方法中，而是在subscribeActual方法中实现了订阅关系。

## 简单的线程切换

下面的这段代码实现了最简单的Rxjava线程切换。发送事件就可以在非UI线程 (RxNewThreadScheduler 的线程执行，将耗时操作放在子线程中，避免阻塞UI线程。接受事件又切换回了Android UI线程，Android禁止在非UI线程操作UI。这样就简单的实现了在主线程处理耗时操作然后在UI线程刷新UI的逻辑。

为了说明问题，把代码拆分成五段，可以看出，其实每次的链式调用都会生成不同的Observable对象，所以我们在平时开发的时候，应该尽可能避免长链式的调用，规避掉不需要的中间操作。

```
private void rxjavaDemo() { Observable observable
= Observable.create(new ObservableOnSubscribe() {
 @Override public void
subscribe(ObservableEmitter e) throws Exception {
 Log.e("rxjava","ObservableEmitter current thread
:"+ Thread.currentThread().getName());
 e.onNext("R"); e.onNext("X");
 e.onComplete(); } });
observable observableSubscribeOn =
observable.subscribeOn(Schedulers.newThread());
observer observer = new Observer() { @Override
 public void onSubscribe(Disposable d) {
 Log.e("rxjava","onSubscribe current thread :" +
Thread.currentThread().getName()); }
@Override public void onNext(Object s) {
 Log.e("rxjava","onNext current thread :" +
Thread.currentThread().getName());
Log.e(RxjavaDemoActivity.class.getSimpleName(), "object :
" + s); } @Override
public void onError(Throwable e) { }
@Override public void onComplete() {
} }; Observable observableObserveOn =
observableSubscribeOn.observeOn(AndroidSchedulers.mainThr
ead());
observableObserveOn.subscribe(observer); }
```

## subscribeOn

下面我们将结合上面的分析和代码，分析一下在  
observable.subscribeOn内部都做了那些操作。

- (1) 实例化了ObservableSubscribeOn对象。并且传入的两个参数分别是ObservableCreate对象和Scheduler对象。
- (2) ObservableCreate对象是我们上面分析的Observable.create生成的一个被观察者。
- (3) Scheduler对象，现在猜测应该是和线程调度有关的类。接下来会分析到。

```
public final Observable<T> subscribeOn(Scheduler scheduler) {
 ObjectHelper.requireNonNull(scheduler, "scheduler is null");
 return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this, scheduler));
}
```

下面在来分析ObservableSubscribeOn类。

```
public final class ObservableSubscribeOn<T> extends AbstractObservableWithUpstream<T, T> {
 final Scheduler scheduler;
 public ObservableSubscribeOn(ObservableSource<T> source, Scheduler scheduler) {
 super(source);
 this.scheduler = scheduler; }
 @Override
 public void subscribeActual(final Observer<? super T> s) {
 final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s);
 s.onSubscribe(parent);
 parent.setDisposable(scheduler.scheduleDirect(new Runnable() {
 @Override
 public void run() {
 source.subscribe(parent);
 }
 }));

 }
}
```

(1)继承了AbstractObservableWithUpstream类。

AbstractObservableWithUpstream类内部存储了ObservableSource类对象，根据上下文，这里的source就是ObservableCreate类对象。

(2)实现了subscribeActual方法，并且在subscribeActual方法内部创建了SubscribeOnObserver对象，这里的SubscribeOnObserver其实也是个观察者，可以理解成事件经过SubscribeOnObserver观察者中转了才最终到达我们创建的观察者。SubscribeOnObserver是AtomicReference的子类(保证原子性)，实现了Observer接口和Disposable接口。

(3)内部调用onSubscribe方法将SubscribeOnObserver传递给观察者，这样观察者就可以控制事件的接受了，即获取了事件发送(被观察者发送数据)的控制权。

(3)ObservableSubscribeOn和ObservableCreate一样，也是Observable的一个子类。并且在ObservableSubscribeOn内部持有它上一步的被观察者Observable的引用(这里就是ObservableCreate)。

(4) source.subscribe(parent)实现了观察者和被观察者之间的订阅关系，并将通过SubscribeOnObserver类对象传递给ObservableOnSubscribe的subscribe方法。SubscribeOnObserver类对象就可以实现事件的发送了。

下面在来看一下SubscribeOnObserver类：作用和代码基本同于CreateEmitter类，看上面的CreateEmitter类分析即可。

```
static final class SubscribeOnObserver<T> extends
AtomicReference<Disposable> implements Observer<T>,
Disposable { private static final long
serialVersionUID = 8094547886072529208L; final
Observer<? super T> actual; final
AtomicReference<Disposable> s;
SubscribeOnObserver(Observer<? super T> actual) {
 this.actual = actual; this.s = new
 AtomicReference<Disposable>(); } @Override
 public void onSubscribe(Disposable s) {
DisposableHelper.setOnce(this.s, s); }
@Override public void onNext(T t) {
actual.onNext(t); } @Override public
void onError(Throwable t) { actual.onError(t);
} @Override public void onComplete()
{ actual.onComplete(); }
@Override public void dispose() {
DisposableHelper.dispose(s);
DisposableHelper.dispose(this); } @Override
public boolean isDisposed() {
DisposableHelper.isDisposed(get()); }
setDisposable(Disposable d) {
DisposableHelper.setOnce(this, d); } }
```

经过上面的分析，我们明确了observable.subscribeOn(Schedulers.newThread())创建的被观察者(Observable) 和观察者(Observer) 之间的订阅关系。下面来分析一下subscribeOn是如何实现线程切换的。首先我们思考一下，如果要实现线程切换，肯定要创建子线程。

**问题：子线程是如何创建**

在调用subscribeOn的时候，传入了Scheduler参数，Scheduler翻译过来就是调度者的意识。通过调用Schedulers的新Thread方法，创建子线程（RxNewThreadScheduler）。下面我们将以RxNewThreadScheduler线程为例。看看线程是如何被创建的。

```
observable observableSubscribeOn =
observable.subscribeOn(Schedulers.newThread());
```

在Schedulers类内部定义了newThread静态方法用于生成Scheduler对象。

(1) 其中NEW\_THREAD为默认的生成的一个Scheduler对象。

```
static final Scheduler NEW_THREAD; public static
Scheduler newThread() { return
RxJavaPlugins.onNewThreadScheduler(NEW_THREAD);
} static { NEW_THREAD =
RxJavaPlugins.initNewThreadScheduler(new
Callable<Scheduler>() { @Override
public Scheduler call() throws Exception {
return NewThreadHolder.DEFAULT; } });
}
```

接下来我们就来看看，initNewThreadScheduler() 是如何生成一个Scheduler实例的。

(1) 在initNewThreadScheduler方法中经过一系列的条件判断，最终会执行到call方法（延迟初始化）。

(2) NewThreadHolder.DEFAULT会返回一个NewThreadScheduler对象（单例模式）

```
static final Scheduler DEFAULT =
NewThreadScheduler.instance(); }
```

下面再来看看单例模式（饿汉式）的NewThreadScheduler类，看名字就可以猜测到是线程调度者。

- (1) NewThreadScheduler 继承自 Scheduler 抽象类。
- (2) 通过静态代码块中创建了 RxThreadFactory 线程工厂对象，该类实现了 ThreadFactory 接口，并且在 RxThreadFactory 类的 newThread 方法中创建了优先级为 5 的线程 Thread。
- (3) 在 NewThreadScheduler 的 createWorker() 方法中，创建了 NewThreadWorker 对象。

```
public final class NewThreadscheduler extends Scheduler {
 private static final String THREAD_NAME_PREFIX =
"RxNewThreadscheduler"; private static final
RxThreadFactory THREAD_FACTORY; private static final
NewThreadscheduler INSTANCE = new NewThreadscheduler();
/** The name of the system property for setting the
thread priority for this Scheduler. */ private static
final String KEY_NEWTREAD_PRIORITY = "rx2.newthread-
priority"; static { int priority =
Math.max(Thread.MIN_PRIORITY,
Math.min(Thread.MAX_PRIORITY,
Integer.getInteger(KEY_NEWTREAD_PRIORITY,
Thread.NORM_PRIORITY))); THREAD_FACTORY = new
RxThreadFactory(THREAD_NAME_PREFIX, priority); }
public static NewThreadscheduler instance() {
 return INSTANCE; } private NewThreadscheduler() {
} @Override public Worker createWorker() {
 return new NewThreadworker(THREAD_FACTORY); }}}
```

接下来我们就来看看 NewThreadWorker 都做了写什么。

- (1) 在 NewThreadWorker 的构造函数中，通过调用 SchedulerPoolFactory.create 的方法并且传入 NewThreadScheduler 中提供的线程工厂 RxThreadFactory 创建了一个 ScheduledExecutorService 对象。

```
public class NewThreadWorker extends Scheduler.Worker
implements Disposable { private final
ScheduledExecutorService executor; volatile boolean
disposed; public NewThreadWorker(ThreadFactory
threadFactory) { executor =
SchedulerPoolFactory.create(threadFactory);
}.....}
```

在来看一下SchedulerPoolFactory类

(1) 通过create方法创建了核心线程数量为1的线程池。

```
/** * Creates a ScheduledExecutorService with the
given factory. * @param factory the thread factory
* @return the ScheduledExecutorService */
public static ScheduledExecutorService create(ThreadFactory
factory) { final ScheduledExecutorService exec =
Executors.newScheduledThreadPool(1, factory); if
(exec instanceof ScheduledThreadPoolExecutor) {
ScheduledThreadPoolExecutor e =
(ScheduledThreadPoolExecutor) exec;
POOLS.put(e, exec); } return exec; }
```

分析到这里我们明确了，通过Schedulers.newThread()会创建一个核心线程数量为1的线程池。

创建完线程，下面就是启动和运行线程了，并且将事件的发送，放在子线程中进行处理。并且我们都知道调用多次 subscribeOn 指定子线程只有第一次会生效，下面我们将带着这两个疑问，来分析一下。

在ObservableSubscribeOn的subscribeActual方法中，通过source.subscribe(parent)调用实现了观察者和被观察者之间的订阅关系。我们可以看到该方法的执行是放在了Runnable里面执行的。所以线程的切换，应该就发生于此处。

(1) 经过上面的分析，这里的scheduler对象是NewThreadScheduler类。并且调用了Schedule的scheduleDirect方法。

```
parent.setDisposable(scheduler.scheduleDirect(new
Runnable() { @Override public void
run() { source.subscribe(parent);
} }));
```

下面来看看Scheduler类的scheduleDirect方法。

- (1) 内部调用了重载的scheduleDirect方法。
- (2) createWorker返回的是NewThreadWorker类对象。并且调用了NewThreadWorker类的schedule方法。

```
public Disposable scheduleDirect(Runnable run) {
return scheduleDirect(run, 0L, TimeUnit.NANOSECONDS);
}
public Disposable scheduleDirect(Runnable run, long
delay, TimeUnit unit) { final Worker w =
createWorker(); final Runnable decoratedRun =
RxJavaPlugins.onSchedule(run); w.schedule(new
Runnable() { @Override public void
run() { try {
decoratedRun.run(); } finally {
w.dispose(); } }
}, delay, unit); return w; }
```

下面来看看NewThreadWorker类的schedule方法。

- (1) 在schedulerActual方法中，通过ScheduledExecutorService执行submit或schedule执行一个Runnable任务，即开启了线程池里面的线程任务。

```

@Override public Disposable schedule(final Runnable run) { return schedule(run, 0, null); }
@Override public Disposable schedule(final Runnable action, long delayTime, TimeUnit unit) {
 if (disposed) { return EmptyDisposable.INSTANCE; }
 return scheduleActual(action, delayTime, unit, null); }
public ScheduledRunnable scheduleActual(final Runnable run, long delayTime,
TimeUnit unit, DisposableContainer parent) {
Runnable decoratedRun = RxJavaPlugins.onSchedule(run);
ScheduledRunnable sr = new
ScheduledRunnable(decoratedRun, parent);
if (parent != null) { if (!parent.add(sr)) {
 return sr; } }
Future<?> f; try { if (delayTime <= 0)
{ f =
executor.submit((Callable<Object>)sr); } else
{ f =
executor.schedule((Callable<Object>)sr, delayTime, unit);
} sr.setFuture(f); } catch
(RejectedExecutionException ex) {
parent.remove(sr); RxJavaPlugins.onError(ex);
} return sr; }

```

分析到这里我们知道了线程的开启是在NewThreadWorker类中进行的。

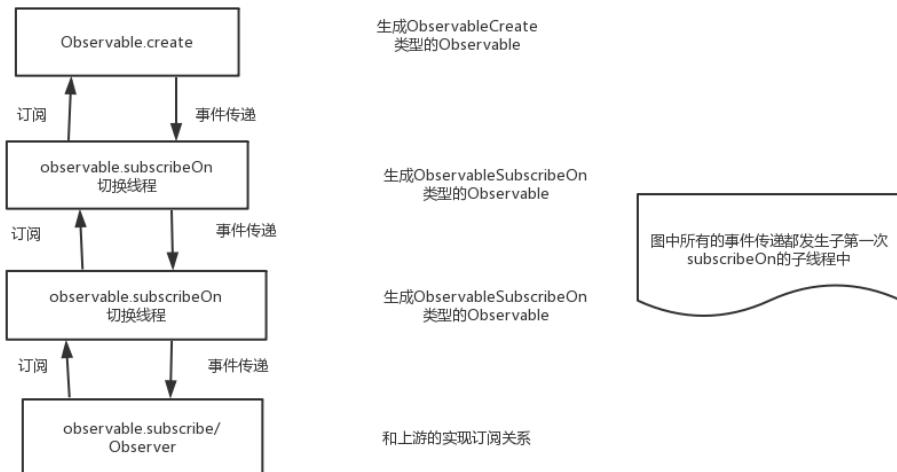
那么还有个疑问：调用多次 subscribeOn 指定子线程只有第一次会生效

- (1) 这里的生效并不是指其他的subscribeOn方法创建的线程没有生效，而是会被第一次的subscribeOn创建的线程“掩盖掉”
- (2) 多次调用subscribeOn，会生成若干个Observable对象，每个新生成的对象都有切换线程的能力，但是只有第一次的subscribeOn才生效，因为后续的线程切换被第一个“掩盖掉”了。

这么说可能有点抽象，下面以一张图来说明：

- (1) 每次调用subscribeOn方法，都会生成一个Observable，并且回持有上游的Observable对象。
- (2) 事件的发送是在第一次的subscribeOn创建的子线程中发送的，中

间不会切换线程。



<https://blog.csdn.net/u011060103>

## observeOn

通过上面subscribeOn的流程梳理，我们知道了上游事件是被如何切换到子线程的。下面我们将分析事件是如何被切换到下游线程的，大部分情况下就是我们的Android UI线程。

下面我们将结合上面的分析和代码，分析一下在observable.observeOn内部都做了那些操作。

```
observable.subscribeOn.observeOn(AndroidSchedulers.mainThread())
```

下面来看一下Observable的observeOn方法：

(1) observeOn 方法返回了一个 ObservableObserveOn类对象。

```
public final Observable<T> observeOn(Scheduler scheduler) { return observeOn(scheduler, false, bufferSize()); } public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bufferSize) { ObjectHelper.requireNonNull(scheduler, "scheduler is null"); ObjectHelper.verifyPositive(bufferSize, "bufferSize"); return RxJavaPlugins.onAssembly(new ObservableobserveOn<T>(this, scheduler, delayError, bufferSize)); }
```

接下来看看 ObservableObserveOn类。

- (1)该类ObservableSubscribeOn基本一致，继承了 AbstractObservableWithUpstream，拥有ObservableSource类型对象，这里是ObservableSubscribeOn实例对象。
- (2)在subscribeActual 方法内部，scheduler是HandlerScheduler类型对象，这里的scheduler就是展开分析了，基本上就是利用Android的 Handler机制实现线程切换的。
- (3)通过 scheduler.createWorker() 创建了 HandlerWorker的Worker对象。
- (4)创建了一个ObserveOnObserver对象，该类实现了Observer 接口，所有它是个Observer，同时实现了一个Runnable接口，这样通过 Handler就可以执行到ObserveOnObserver的run方法。

```
public final class ObservableobserveOn<T> extends AbstractObservableWithUpstream<T, T> { final Scheduler scheduler; final boolean delayError; final int bufferSize; public ObservableobserveOn(ObservableSource<T> source, Scheduler scheduler, boolean delayError, int bufferSize) { super(source); this.scheduler = scheduler; this.delayError = delayError; this.bufferSize = bufferSize; } @Override protected void subscribeActual(Observer<? super T> observer) { if (scheduler instanceof TrampolineScheduler) { source.subscribe(observer); } else { Scheduler.Worker w = scheduler.createWorker(); source.subscribe(new ObserveOnObserver<T>(observer, w, delayError, bufferSize)); } } }
```

下面就来看看这个 `ObserveOnObserver`, 通过上面我们知道, 线程切换是通过 `Handler` 实现的。

- (1) `actual` 参数是我们创建的 `Observer` 对象。
- (2) `Worker` 参数是 `HandlerWorker` 对象。通过 `AndroidSchedulers.mainThread()` 的调用是创建的。

```
static final class ObserveOnObserver<T> extends BasicIntQueueDisposable<T> implements Observer<T>, Runnable { private static final long serialVersionUID = 6576896619930983584L; final Observer<? super T> actual; final Scheduler.Worker worker; final boolean delayError; final int bufferSize; SimpleQueue<T> queue; Disposable s; Throwable error; volatile boolean done; volatile boolean cancelled; int sourceMode; boolean outputFused; ObserveOnObserver(Observer<? super T> actual, Scheduler.Worker worker, boolean delayError, int bufferSize) { this.actual = actual; this.worker = worker; this.delayError = delayError; this.bufferSize = bufferSize; } }
```

下面来看一下run方法里面的操作：

outputFused参数默认是false，所以接下来看看drainNormal方法。

```
@Override public void run() { if (outputFused) { drainFused(); } else { drainNormal(); } }
```

(1) queue参数是在onSubscribe方法里面创建的。而onSubscribe方法的调用，则是在上游的subscribeActual方法中调用的。

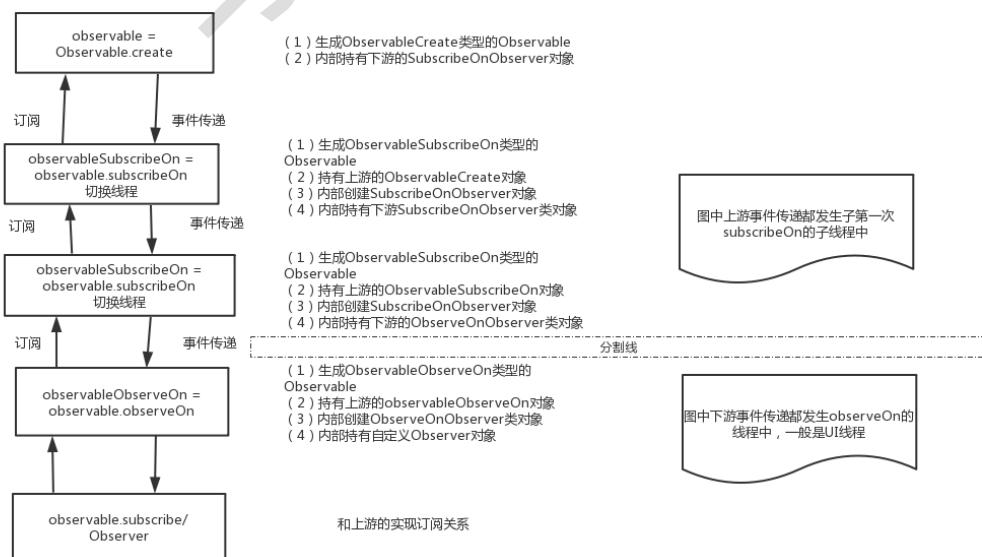
(2) 内部通过轮训队列里面的事件，将事件最终发送到Observer。

```

void drainNormal() {
 final SimpleQueue<T> q = queue;
 final Observer<? super T> a = actual;
 int missed = 1;
 for (;;) {
 if (checkTerminated(done, q.isEmpty(), a)) {
 return;
 }
 for (;;) {
 boolean d = done;
 try {
 T v;
 v = q.poll();
 } catch (Throwable ex) {
 Exceptions.throwIfFatal(ex);
 s.dispose();
 a.onError(ex);
 }
 if (empty = v == null) {
 if (checkTerminated(d, empty, a)) {
 return;
 }
 if (empty) {
 break;
 }
 a.onNext(v);
 }
 missed = addAndGet(-missed);
 if (missed == 0) {
 break;
 }
 }
 }
}

```

下面以一张图，总结一下：observeOn和subscribeOn的流程。



- (1)subscribeOn 控制上游线程切换， subscribeOn多次调用只有第一次的 subscribeOn会起作用。
- (2)observeOn控制下游线程切换。 observeOn可以使用多次。 并且 observeOn 后面的所有操作都会在observeOn指定的线程中执行。
- (3)subscribeOn和observeOn之间的操作， 会在subscribeOn 指定的线程中执行， 直到执行了observeOn操作。

### 20.2.3 RxJava2线程切换原理分析

#### 一、概述

本节将分析RxJava2的线程切换模型。通过对线程切换源代码的分析到达对RxJava2线程切换彻底理解的目的。通过对本节的学习你会发现，RxJava2线程切换是如此的简单，仅仅是通过两个操作符就能完成从子线程到主线程，或者主线程到子线程，再或者从子线程到子线程的切换。对应的操作符为： observerOn：指定观察者运行的线程。 subscribeOn：执行被观察者运行的线程。

#### 二、简单例子入手

```
private void threadSwitchTest() {
 Observable<String> observable = observable.create(new
 ObservableOnSubscribe<String>() {
 @Override
 public void subscribe(ObservableEmitter<String>
 emitter) throws Exception {
 emitter.onNext("《深入Java虚拟机》");
 MyLog.log("Thread:" + Thread.currentThread().getName());
 }
 });
 observable
.observeOn(AndroidSchedulers.mainThread())//观察者执行线程
.subscribeOn(Schedulers.io())//被观察者执行线程
.subscribe(new Consumer<String>() {
 @Override
 public void
 accept(String s) throws Exception {
 MyLog.log("Thread:" + Thread.currentThread().getName());
 }
});
}
```

以上例子中我们使用observeOn(AndroidSchedulers.mainThread())来指定观察者运行在主线程，使用subscribeOn(Schedulers.io())来指定被观察运行在子线程

### 三、源码分析

本节针对RxJava2的源代码我们需要弄明白三件事情：

- 1.子线程如何切换到主线程原理分析
- 2.主线程如何切换到子线程原理分析
- 3.子线程如何切换到子线程原理分析

通过上一节的分析我们知道RxJava2通过创建一个被观察者(ObservableCreate)和一个观察者(LambdaObserver)，并实现观察者和被观察者的绑定。通过ObservableEmitter.onNext发送消息，Consumer.accept中接收消息。而操作符map仅仅是对被观察者ObservableCreate做了一层包装（装饰模式），变成了ObservableMap。而观察者装饰后则变成了MapObserver。

很显然，observeOn和subscribeOn都属于操作符（他们都是用来做线程切换的操作符而已），所以这两个操作符也符合上面Map操作符的包装规则。

subscribeOn源码分析：

```
@CheckReturnValue
@schedulerSupport(SchedulerSupport.CUSTOM) public
final Observable<T> subscribeOn(Scheduler scheduler) {
 ObjectHelper.requireNonNull(scheduler, "scheduler is
 null"); return RxJavaPlugins.onAssembly(new
 ObservableSubscribeOn<T>(this, scheduler)); }
```

从上述源码可以看出subscribeOn确实如上面所说，会被包装成为一个ObservableSubscribeOn。其构造方法会传入两个参数，一个是this：代表当前被观察者，也就是操作符上面修饰的那个被观察者，本例中指的是ObservableObserveOn,ObservableObserverOn又装饰了ObservableCreate。scheduler指的是Schedulers.io(),指被观察者运行在io线程，也就是子线程中。

下面看下Schedulers类是个什么东西。

```
public final class Schedulers { @NotNull static
final Scheduler SINGLE; @NotNull static final
Scheduler COMPUTATION; @NotNull static final
Scheduler IO; @NotNull static final Scheduler
TRAMPOLINE; @NotNull static final Scheduler
NEW_THREAD; static final class SingleHolder {
static final Scheduler DEFAULT = new SingleScheduler();
} static final class ComputationHolder { static
final Scheduler DEFAULT = new ComputationScheduler();
} static final class IoHolder { static final
Scheduler DEFAULT = new IoScheduler(); } static
final class NewThreadHolder { static final
Scheduler DEFAULT = new NewThreadsScheduler(); }
static { SINGLE =
RxJavaPlugins.initSingleScheduler(new SingleTask());
COMPUTATION =
RxJavaPlugins.initComputationScheduler(new
ComputationTask()); IO =
RxJavaPlugins.initIOScheduler(new IOTask());
TRAMPOLINE = TrampolineScheduler.instance();
NEW_THREAD = RxJavaPlugins.initNewThreadsScheduler(new
NewThreadTask()); }
```

Schedulers内部封装了各种Scheduler。每一个Scheduler中都封装的有线程池，用于执行后台任务。

到此处ObservableSubscribeOn对象也就创建完成了。

下面看下ObserverOn操作符都干了什么事情：

```
@CheckReturnValue
@schedulersSupport(SchedulersSupport.CUSTOM) public
final Observable<T> observeOn(Schedulers scheduler,
boolean delayError, int bufferSize) {
ObjectHelper.requireNonNull(scheduler, "scheduler is
null"); ObjectHelper.verifyPositive(bufferSize,
"bufferSize"); return RxJavaPlugins.onAssembly(new
ObservableobserveOn<T>(this, scheduler, delayError,
bufferSize)); }
```

ObserveOn方法内部包装了一个ObservableObserveOn对象，其有两个参数，this：代表当前Observable对象，此处指的是ObservableCreate这个对象，scheduler代表的是AndroidSchedulers.mainThread()。

我们看一下AndroidSchedulers的源代码，看它都干了写什么事

```
public final class AndroidSchedulers { private static
final class MainHolder { static final Scheduler
DEFAULT = new HandlerScheduler(new
Handler(Looper.getMainLooper()), false); } private
static final Scheduler MAIN_THREAD =
RxAndroidPlugins.initMainThreadScheduler(new
Callable<Scheduler>() { @Override public
Scheduler call() throws Exception {
return MainHolder.DEFAULT; }
}); /* * A {@link Scheduler} which executes actions
on the Android main thread. */ public static
Scheduler mainThread() { return
RxAndroidPlugins.onMainThreadScheduler(MAIN_THREAD);
} /* * A {@link Scheduler} which executes actions on
{@code looper}. */ public static Scheduler
from(Looper looper) { return from(looper, false);
}
```

AndroidSchedulers的内部类MainHolder的作用是在主线程中创建一个Handler。由new Handler(Looper.getMainLooper())来完成。因为Looper所在的线程为Handler所在的线程，又因为Looper.getMainLooper()获取到的是主线程的looper，所以当前Handler

运行在主线程，顺带着这块的逻辑也是在主线程中完成的。字段 MAIN\_THREAD仅仅是把HandlerScheduler返回而已，而 HandlerScheduler持有主线程handler。那么manThread()方法就好理解了，就是返回了一个持有主线程Handler的Scheduler而已。

所以ObservableObserverOn包装了ObservableCreate并持有了主线程 Handler。到此被观察者就已经创建完成了。

下面说下观察者Consumer.accept方法在这个链式调用中是如何被执行的：

1. 经过上面的分析被观察者已经变为： ObservableObserverOn， ObservableObserverOn持有ObservableSubscribeOn对象的引用， ObservableSubscribeOn又持有ObservableCreate的引用。所以 Observable对象经过三层包装最终成为了ObservableObserverOn。

2. Observable.subscribe(Consumer consumer)方法执行订阅，会把原始的观察者对象LambdaObserver对象包装成为 ObserverOnObserver对象, ObserverOnObserver又会被包装成 SubscribeOnObserver对象。用以在ObservableSubscribeOn对象执行 subscribeActual方法的时候正式执行绑定操作。至此，观察者和被观察者建立了绑定关系。

```
public final class ObservableSubscribeOn<T> extends AbstractObservableWithUpstream<T, T> { final Scheduler scheduler; public ObservableSubscribeOn(ObservableSource<T> source, Scheduler scheduler) { super(source); this.scheduler = scheduler; } @Override public void subscribeActual(final Observer<? super T> s) { final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s); s.onSubscribe(parent); parent.setDisposable(scheduler.scheduleDirect(new SubscribeTask(parent))); }}
```

从上面的代码中我们基本无法判断是在哪里绑定的。从上面的分析我们知道scheduler要是一个HandlerScheduler.那么我们可以断定的是scheduler.scheduleDirect一定是用来执行任务的，那么SubscribeTask肯定是一个任务没错。事实也如我们所料一样，确实是这样的。

### 看下HandlerScheduler的scheduleDirect都干了什么

```
@Override public Disposable scheduleDirect(Runnable run, long delay, TimeUnit unit) { if (run == null) throw new NullPointerException("run == null"); if (unit == null) throw new NullPointerException("unit == null"); run = RxJavaPlugins.onSchedule(run); ScheduledRunnable scheduled = new ScheduledRunnable(handler, run); handler.postDelayed(scheduled, unit.toMillis(delay)); return scheduled; }
```

非常的简单，构建一个ScheduleRunnable，并把handler和runnable传入进去，然后执行handler.postDelayed向handler发送消息就行了。postDelayed方法最终会调用Runnable的run方法。

```
private static final class ScheduledRunnable implements Runnable, Disposable { private final Handler handler; private final Runnable delegate; private volatile boolean disposed; // Tracked solely for isDisposed(). ScheduledRunnable(Handler handler, Runnable delegate) { this.handler = handler; this.delegate = delegate; } @Override public void run() { try { delegate.run(); } catch (Throwable t) { RxJavaPlugins.onError(t); } } void dispose() { disposed = true; } boolean isDisposed() { return disposed; } }
```

ScheduleRunnable在run方法中又会调用SubscribeTask的run方法。

SubscribeTask.java

```
final class SubscribeTask implements Runnable {
 private final SubscribeOnObserver<T> parent;
 SubscribeTask(SubscribeOnObserver<T> parent) {
 this.parent = parent; } @Override
 public void run() { source.subscribe(parent);
 }}
```

在subscribeTask的run方法中最终完成了绑定，source指 ObservableOnSubscribe

3.被观察者在执行ObservableOnSubscribe实例的subscribe方法的 ObservableEmitter参数的onNext方法的时候,会首先调用 SubscribeOnObserver的onNext方法，又由于SubscribeOnObserver持有ObserverOnObserver的引用，因此在SubscribeOnObserver的 onNext方法中又会调用ObserveOnObserver对象的onNext方法，在此 Next方法中又会调用CreateObserver的onNext方法,在其内部又会调用 LambdaObserver.onNext，然后在LambdaObserver的onNext方法中 又会调用Consumer.accept方法。最后完成数据的从发送到接收的流转。

了解了以上操作符的整体流转流程后，我们接下面回过头来看开头我们提出的三个问题：

### 1.主线程切换到子线程

我们先来看ObservableSubscribeOn这个类，在上面的小例子中，直接将被观察者运行在IO线程中了。我们直接看 ObservableSubscribeOn的subscribeActual方法的源代码

```
@Override public void subscribeActual(final
 Observer<? super T> s) { final
 SubscribeOnObserver<T> parent = new
 SubscribeOnObserver<T>(s); s.onSubscribe(parent);
 parent.setDisposable(scheduler.scheduleDirect(new
 SubscribeTask(parent))); }
```

在subscribeActual方法内部先创建一个SubscribeOnObserver对象，并执行setDisposable执行任务。这里的scheduler指的是HandlerScheduler。SubscribeTask是一个实现了Runnable的对象在其内部完成了绑定操作。

先来看下HandlerScheduler的scheduleDirect方法

```
@Override public Disposable scheduleDirect(Runnable run, long delay, TimeUnit unit) { if (run == null) throw new NullPointerException("run == null"); if (unit == null) throw new NullPointerException("unit == null"); run = RxJavaPlugins.onSchedule(run); ScheduledRunnable scheduled = new ScheduledRunnable(handler, run); handler.postDelayed(scheduled, unit.toMillis(delay)); return scheduled; }
```

scheduleDirect方法逻辑上很简单，1.把SubscribeTask和handler封装成ScheduledRunnable。然后利用Handler.postDelayed执行这个Runnable对象。postDelayed执行的最后会调用msg.callback.run()其实就是调用ScheduledRunnable的run方法。在run方法内又会调用SubscribeTask的run方法。

```
private static final class ScheduledRunnable implements Runnable, Disposable { private final Handler handler; private final Runnable delegate; private volatile boolean disposed; // Tracked solely for isDisposed(). ScheduledRunnable(Handler handler, Runnable delegate) { this.handler = handler; this.delegate = delegate; } @Override public void run() { try { //SubscribeTask的run方法 delegate.run(); } catch (Throwable t) { RxJavaPlugins.onError(t); } }}
```

我们看下SubscribeTask的run方法都干了啥事。

```
final class SubscribeTask implements Runnable {
 private final SubscribeOnObserver<T> parent;
 SubscribeTask(SubscribeOnObserver<T> parent) {
 this.parent = parent; } @Override
 public void run() { source.subscribe(parent);
 }}
```

在SubscribeTask类的run方法中完成最终的绑定。此处的source指的是ObservableOnSubscribe

在主线程中执行其实也就这么多最终会把方法放到Handler中执行

## 2.在子线程中执行任务

直接看ObservableObserveOn类的subscribeActual

```
@Override protected void subscribeActual(Observer<?
super T> observer) { //首先判断一下调度线程是否是在当前
线程中执行，如果是就直接绑定，如果不是就开启工作线程 if
(scheduler instanceof TrampolineScheduler) {
 source.subscribe(observer); } else {
 scheduler.worker w = scheduler.createWorker();
 source.subscribe(new ObserveOnObserver<T>(observer, w,
delayError, bufferSize)); } }
```

首先根据TrampolineScheduler判断任务是否是在当前线程执行，如果是就直接绑定。如果不是就创建一个ObserverOnObserver对象，并把Observer和Worker对象传递进去。即可完成绑定。

我们接下来主要看下其是如何在子线程中执行的

ObserverOnObserver继承了Runnable对象。在执行onNext方法的时候会调用worker的schedule(this)方法。

```
@Override public void onNext(T t) { if
(done) { return; }
if (sourceMode != QueueDisposable.ASYNC) {
queue.offer(t); } schedule();
}
```

```
void schedule() { if (getAndIncrement() == 0) { worker.schedule(this); } }
```

其实到这我们大致可以判断出来worker.schedule(this)必定会运行run方法。不着急，我们先看IoSchedule类中的worker以及worker.schedule干了什么

```
@NotNull @Override public Worker createWorker() { return new EventLoopWorker(pool.get()); }
```

创建一个Worker对象

```
@NotNull @Override public Disposable schedule(@NotNull Runnable action, long delayTime, @NotNull TimeUnit unit) { if (tasks.isDisposed()) { // don't schedule, we are unsubscribed return EmptyDisposable.INSTANCE; } return threadWorker.scheduleActual(action, delayTime, unit, tasks); }
```

执行threadWorker.scheduleActual(action),这里的action指的就是ObserverObserveOn对象，因其继承了Runnable对象。

看看ThreadWorker.scheduleActual干了啥

```

@NonNull public ScheduledRunnable
scheduleActual(final Runnable run, long delayTime,
@NonNull TimeUnit unit, @Nullable DisposableContainer
parent) { Runnable decoratedRun =
RxJavaPlugins.onSchedule(run); ScheduledRunnable
sr = new ScheduledRunnable(decoratedRun, parent);
if (parent != null) { if (!parent.add(sr)) {
return sr; }
}
Future<?> f; try { if (delayTime <= 0)
{
f = executor.submit((Callable<Object>)sr); } else
{
f = executor.schedule((Callable<Object>)sr, delayTime, unit);
}
sr.setFuture(f); } catch
(RejectedExecutionException ex) { if (parent
!= null) { parent.remove(sr);
}
RxJavaPlugins.onError(ex);
}
return sr; }

```

逻辑很清晰，把传入的Runnable (ObservableObserveOn) 封装成一个ScheduleRunnable对象。并把这个对象放入线程池中去执行。

executor都代表线程池。执行的时候会运行ScheduleRunnable的run方法。在其run方法内部又会调用ObserverObserveOn的run方法。

下面回过头来再看看ObserverObserveOn的run方法

```

@Override public void run() { if
(outputFused) { drainFused(); }
else { drainNormal(); }
}

```

```

void drainNormal() {
a.onNext(v);
}

```

其会调用a.onNext方法，让onNext方法运行在线程池中。a值的就是一个CreateObserver或者其包装类。通过一层层的调用Consume.accept方法最终会运行到子线程中。

## 2.主线程如何切换到主线程

回过头看ObservableObserveOn的subscribeActual方法

```
 @Override
 protected void subscribeActual(Observer<? super T> observer) {
 if (scheduler instanceof TrampolineScheduler) {
 source.subscribe(observer);
 } else {
 Scheduler.Worker w = scheduler.createWorker();
 source.subscribe(new ObserveOnObserver<T>(observer, w, delayError, bufferSize));
 }
 }
```

这里的scheduler指的是HandlerScheduler。HandlerScheduler内部维护了一个运行在主线程的Handler和一个内部类HandlerWorker。其调用source.subscribe执行观察者和被观察者的订阅。当ObservableEmitter.onNext方法执行后，会调用ObserveOnObserver内部的onNext方法。

```
@Override
public void onNext(T t) {
 if (done) {
 return;
 }

 if (sourceMode != QueueDisposable.ASYNC) {
 queue.offer(t);
 }
 schedule();
}
```

schedule方法又会调用worker.scheduler方法

```
void schedule() {
 if (getAndIncrement() == 0) {
 worker.schedule(run: this);
 }
}
```

此处的worker为HandlerScheduler中的Worker，源码如下

```
private static final class HandlerWorker extends Worker {
 private final Handler handler;
 private final boolean async;

 private volatile boolean disposed;

 HandlerWorker(Handler handler, boolean async) {
 this.handler = handler;
 this.async = async;
 }

 @Override
 @SuppressLint("NewApi") // Async will only be true when the API is available to call.
 public Disposable schedule(Runnable run, long delay, TimeUnit unit) {
 if (run == null) throw new NullPointerException("run == null");
 if (unit == null) throw new NullPointerException("unit == null");

 if (disposed) [
 return Disposables.disposed();
]

 run = RxJavaPlugins.onSchedule(run);

 ScheduledRunnable scheduled = new ScheduledRunnable(handler, run);

 Message message = Message.obtain(handler, scheduled);
 message.obj = this; // Used as token for batch disposal of this worker's runnables.

 if (async) {
 message.setAsynchronous(true);
 }

 handler.sendMessageDelayed(message, unit.toMillis(delay));
 }
}
```

通过Handler把ScheduleRunnable发送到主线程中执行。因为HandlerScheduler是主线程handler所以在Handler中执行的逻辑也会被切换到主线程中去执行。其实这里的run方法最终运行的是ObserveOnObserver中的run方法。在其run方法中会调用其上级包装类SubscribeOnObserver的onNext方法。之后又会调用LambdaObserver的onNext方法。在其onNext方法中会调用Consumer.accept方法，最终让其运行在主线程中。

### 3.子线程如何切换到子线程

这里分析下把Consumer.accept方法运行在子线程的流程

同样只需要设置observeOn(Schedulers.io())就OK了。同样会创建一个ObserveOnObserver，其接受两个重要的参数this：当前Observer，scheduler：ioScheduler。

其绑定过程会执行ObservableObserveOn的subscribeActual方法

```

@Override
protected void subscribeActual(Observer<? super T> observer) {
 if (scheduler instanceof TrampolineScheduler) {
 source.subscribe(observer);
 } else {
 Scheduler.Worker w = scheduler.createWorker();
 source.subscribe(new ObserveOnObserver<T>(observer, w, delayError, bufferSize));
 }
}

```

只是此处的scheduler不再是HandlerScheduler，而是IoScheduler。当 ObservableEmitter.onNext方法被执行的时候，会调用 ObserveOnObserver的onNext方法。而在onNext方法中又会调用 IoScheduler中worker.schedule。最终会执行NewThreadWorker的 scheduleActual方法

```

@NonNull
public ScheduledRunnable scheduleActual(final Runnable run, long delayTime, @NonNull TimeUnit unit, @Nullable Disposable decoratedRun = RxJavaPlugins.onSchedule(run);

ScheduledRunnable sr = new ScheduledRunnable(decoratedRun, parent);

if (parent != null) {
 if (!parent.add(sr)) {
 return sr;
 }
}

Future<?> f;
try {
 if (delayTime <= 0) {
 f = executor.submit((Callable<Object>)sr);
 } else {
 f = executor.schedule((Callable<Object>)sr, delayTime, unit);
 }
 sr.setFuture(f);
} catch (RejectedExecutionException ex) {
 if (parent != null) {
 parent.remove(sr);
 }
 RxJavaPlugins.onError(ex);
}

return sr;
}

```

当上述方法被执行后就会调用ObserveOnObserver中的run方法。其run方法又会逐个解包装调用其OnNext方法。知道LambdaObserver的 onNext被调用。onNext又会调用Consumer.accept。经过以上步骤就完成了最终的调用。因为run是在线程池中执行的，所以跟着把业务逻辑代码也切换到了线程池中执行，即子线程中执行。

总结：

经过上面的分析，RxJava切换线程已经分析完了，相信大家了解后对 RxJava的线程切换会有一定的感悟。在这里再用白花总结一下。

- 1.子线程切换主线程：给主线程所在的Handler发消息，然后就把逻辑切换过去了。

2.主线程切换子线程：把任务放到线程池中执行就能把执行逻辑切换到子线程

3.子线程切换子线程：把任务分别扔进两个线程就行了。

## 20.3 Rxjava内存泄漏防止方案——RxLifecycle, AutoDispose, RxLife框架

### 20.3.1 Android 使用RxLifecycle解决RxJava内存泄漏

[RxLifecycle GitHub地址](#)

[RxJava GitHub地址](#)

#### 1.为什么会发生内存泄漏

使用RxJava发布一个订阅后，当页面被finish，此时订阅逻辑还未完成，如果没有及时取消订阅，就会导致Activity/Fragment无法被回收，从而引发内存泄漏。

写段代码测试一下，定义一个Activity，布局中显示一张图片，这样可以直观的看到此Activity的内存占用情况，然后在Activity中发布一个订阅后，关闭Activity，订阅逻辑如下：

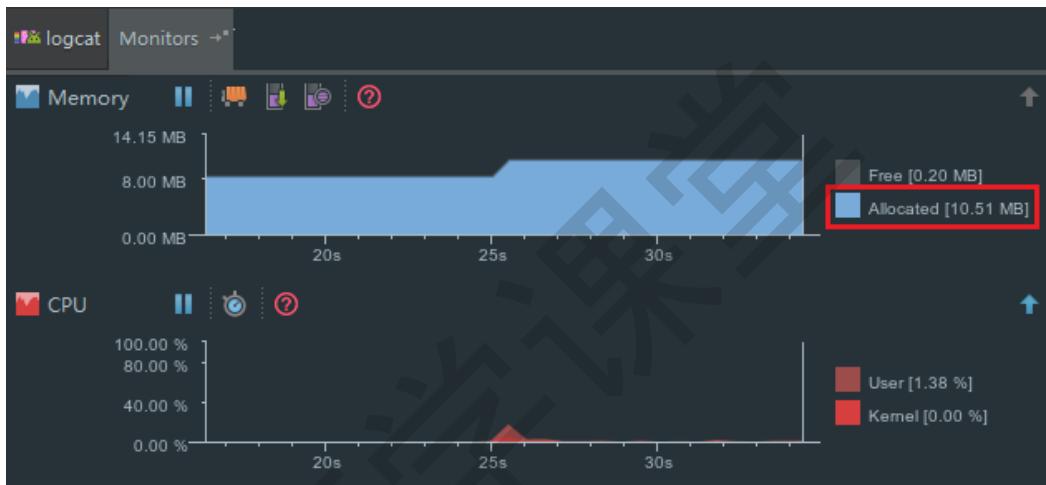
```
// 每隔1s执行一次事件
Observable.interval(1,
TimeUnit.SECONDS)
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())
.subscribe(new Observer<Long>() {
 @Override
 public void onSubscribe(@NonNull Disposable d) {
 }
 @Override
 public void onNext(@NonNull Long aLong) {
 Log.i("接收数据", String.valueOf(aLong));
 }
 @Override
 public void onError(@NonNull
Throwable e) {
 }
 @Override
 public void onComplete() {
 }
});
```

看下打开Activity之前的内存占用情况：



打开Activity之前的内存占用情况

打开Activity之后的内存占用情况：



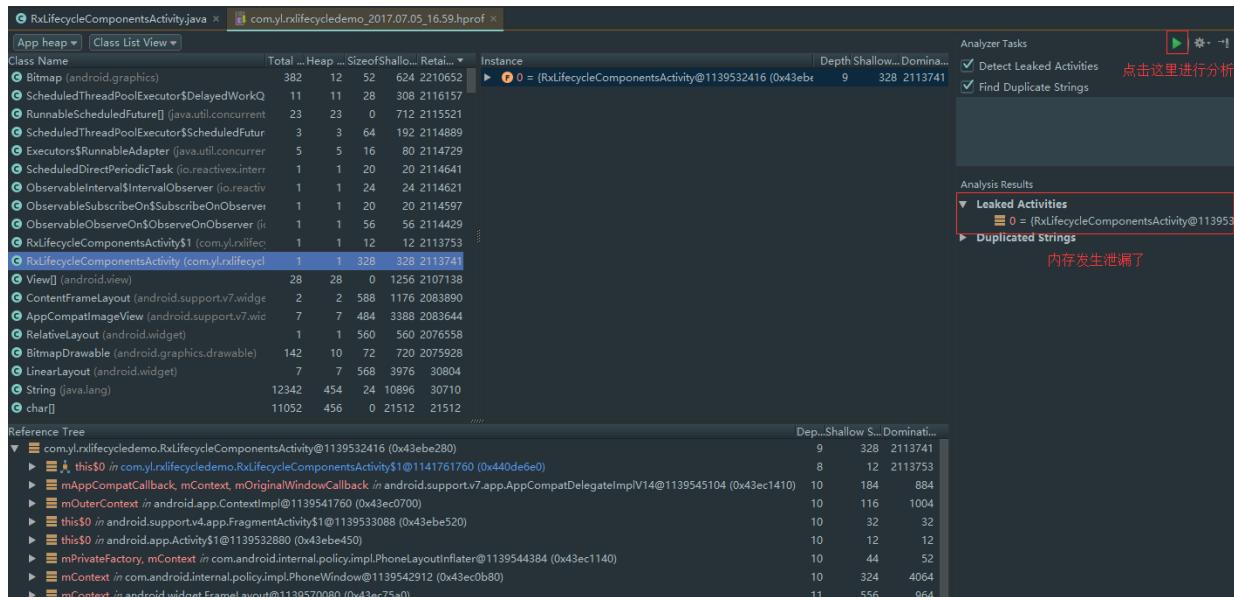
打开Activity之后的内存占用情况

关闭Activity，手动执行GC（点击小车图标），发现内存占用并没有减少：



手动执行GC

导出hprof文件进行分析（点击小车图标右边的图标），发现已经发生了内存泄漏：



## 内存泄漏分析

那么除了在onDestory方法中手动取消订阅之外，还有什么方法可以避免上述的泄漏问题呢，这时RxLifecycle就派上用场了。

## 2.RxLifecycle是什么

看下官方的介绍：

This library allows one to automatically complete sequences based on a second lifecycle stream.

This capability is useful in Android, where incomplete subscriptions can cause memory leaks.

大概意思就是：可以通过绑定生命周期的方式，来解决内存泄漏的问题。

## 3.实践

看下使用RxLifecycle需要依赖的库：

```
// RxLifecycle基础库compile
'com.trello.rxlifecycle2:rxlifecycle:2.1.0'// Android使用的库，里面使用了Android的生命周期方法// 内部引用了基础库，如果使用此库则无需再引用基础库compile
'com.trello.rxlifecycle2:rxlifecycle-android:2.1.0'// Android组件库，里面定义了例如RxAppCompatActivity、RxFragment之类的Android组件// 内部引用了基础库和Android库，如果使用此库则无需再重复引用compile 'com.trello.rxlifecycle2:rxlifecycle-components:2.1.0'// Android使用的库，继承NaviActivity使用compile 'com.trello.rxlifecycle2:rxlifecycle-navi:2.1.0'// Android使用的库，继承LifecycleActivity使用// 需要引入Google的仓库支持，用法和rxlifecycle-navi类似compile
'com.trello.rxlifecycle2:rxlifecycle-android-lifecycle:2.1.0'// Google的仓库支持allprojects {
repositories { jcenter() maven { url
'https://dl.google.com/dl/android/maven2/' } } } // 支持Kotlin语法的RxLifecycle基础库compile
'com.trello.rxlifecycle2:rxlifecycle-kotlin:2.1.0'// 支持Kotlin语法的Android库compile
'com.trello.rxlifecycle2:rxlifecycle-android-lifecycle-kotlin:2.1.0'
```

本文需要依赖其中两个库：

```
// 依赖以下两个库，会自动引用基础库与Android库compile
'com.trello.rxlifecycle2:rxlifecycle-components:2.1.0'compile
'com.trello.rxlifecycle2:rxlifecycle-navi:2.1.0'
```

## rxlifecycle-components

```
public class RxLifecycleComponentsActivity extends RxAppCompatActivity { @Override protected void onCreate(@Nullable Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity_rxlifecycle_components); } @Override protected void onStart() { super.onStart(); Observable.interval(1, TimeUnit.SECONDS) .subscribeOn(Schedulers.io()) .observeOn(AndroidSchedulers.mainThread()) .compose(this.<Long>bindToLifecycle()) .subscribe(); }}
```

使用compose(this.bindToLifecycle())方法绑定Activity的生命周期，在onStart方法中绑定，在onStop方法被调用后就会解除绑定，以此类推。

**有一种特殊情况，如果在onPause/onStop方法中绑定，那么就会在它的下一个生命周期方法（onStop/onDestory）被调用后解除绑定。**

再次运行程序，打开关闭Activity，手动进行GC，发现占用内存减少了，导出hprof文件进行分析，没有发生内存泄漏。

除了使用bindToLifecycle的方式之外，还可以指定取消订阅的时机：

```
public class RxLifecycleComponentsActivity extends RxAppCompatActivity { @Override protected void onCreate(@Nullable Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.activity_rxlifecycle_components); ButterKnife.bind(this); initData(); } private void initData() { Observable.interval(1, TimeUnit.SECONDS) .subscribeOn(Schedulers.io()) .observeOn(AndroidSchedulers.mainThread()) .compose(this. <Long>bindUntilEvent(ActivityEvent.DESTROY)) .subscribe(); }}
```

使用compose(this.bindUntilEvent(ActivityEvent.DESTROY))方法指定在onDestroy方法被调用时取消订阅。

**注意compose方法需要在subscribeOn方法之后使用，因为在测试的过程中发现，将compose方法放在subscribeOn方法之前，如果在被观察者中执行了阻塞方法，比如Thread.sleep()，取消订阅后该阻塞方法不会被中断。**

```
observable<String> observable = observable.create(new
 ObservableOnSubscribe<String>() { @Override
 public void subscribe(@NonNull
 ObservableEmitter<String> e) throws Exception {
 try { Thread.sleep(60 * 1000);
 } catch (InterruptedException ex) {
 ex.printStackTrace();
 }
 });
 observable
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())
 .compose(this.
<String>bindUntilEvent(ActivityEvent.DESTROY))
 .subscribe();
```

rxlifecycle-navi

想  
識  
思  
學

```
public class RxLifecycleNaviActivity extends NaviActivity
{
 private final LifecycleProvider<ActivityEvent>
provider = NaviLifecycle.createActivityLifecycleProvider(this);
@Override protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_rxlifecycle_navi);
ButterKnife.bind(this); initData(); }
private void initData() { Observable.interval(1,
TimeUnit.SECONDS)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.compose(provider.<Long>bindUntilEvent(ActivityEvent.DESTROY))
.subscribe(new Observer<Long>() {
@Override public void
onSubscribe(@NonNull Disposable d) { }
@Override public
void onNext(@NonNull Long aLong) {
Log.i("接收数据", String.valueOf(aLong));
}
@Override public
void onError(@NonNull Throwable e) { }
@Override public
void onComplete() { }
}); } @Override protected void onStart() {
super.onStart(); Observable.interval(1,
TimeUnit.SECONDS)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.compose(provider.<Long>bindToLifecycle())
.subscribe(new Observer<Long>() {
@Override public void
onSubscribe(@NonNull Disposable d) { }
@Override public
void onNext(@NonNull Long aLong) {
Log.i("接收数据", String.valueOf(aLong));
}
@Override public

```

```
void onError(@NonNull Throwable e) {
 @Override
 public void onComplete() {
 }
}
```

继承了NaviActivity，通过  
NaviLifecycle.createActivityLifecycleProvider(this)方法获取  
LifecycleProvider对象，有了LifecycleProvider对象之后就可以调用  
bindToLifecycle或者bindUntilEvent方法了。

**如果你使用的是MVP结构，这个LifecycleProvider对象可以直接传给  
Presenter层使用。**

继承RxAppCompatActivity为什么就能直接用this的方式调用  
bindToLifecycle或bindUntilEvent方法呢？看下源码，原来  
RxAppCompatActivity实现了LifecycleProvider接口。

```
public abstract class RxAppCompatActivity extends
AppCompatActivity implements
LifecycleProvider<ActivityEvent> {}
```

#### 4.写在最后

源码已托管到GitHub上，欢迎Fork，觉得还不错就Start一下吧！

[GitHub传送门](#)

欢迎同学们吐槽评论，如果你觉得本篇博客对你有用，那么就留个言或者  
点下喜欢吧( ^ - ^ )

### 20.3.2 解决RxJava内存泄漏（前篇）：RxLifecycle详解及原理分析

#### 前言

随着RxJava及RxAndroid的逐渐推广，使用者越来越多，但是有一个问题，RxJava的使用不当极有可能会导致内存泄漏。

比如，使用RxJava发布一个订阅后，当Activity被finish，此时订阅逻辑还未完成，如果没有及时取消订阅，就会导致Activity无法被回收，从而引发内存泄漏。

目前网上对RxJava的内存泄漏有几种方案：

- 1、通过封装，手动为RxJava的每一次订阅进行控制，在指定的时机进行取消订阅；
- 2、使用[Daniel Lew](#) 的[RxLifecycle](#)，通过监听Activity、Fragment的生命周期，来自动断开subscription以防止内存泄漏。

笔者上述两种方式都使用过，**RxLifecycle**显然对于第一种方式，更简单直接，并且能够在Activity/Fragment容器的**指定生命周期**取消订阅，实在是好用。

## 依赖并使用RxLifecycle

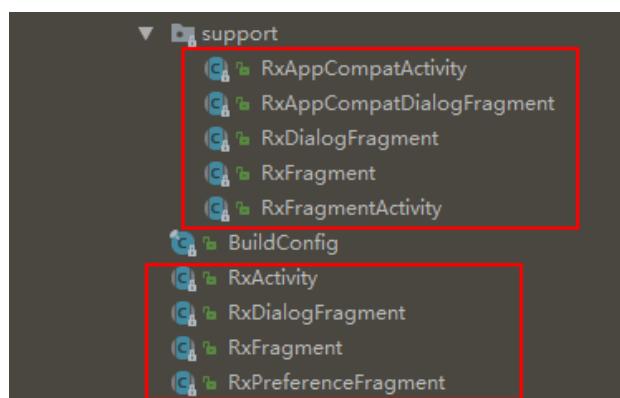
### 1、添加依赖

首先在build.gradle文件中添加依赖：

```
compile
'com.trello.rxlifecycle2:rxlifecycle:2.2.1'
compile
'com.trello.rxlifecycle2:rxlifecycle-
android:2.2.1'
compile
'com.trello.rxlifecycle2:rxlifecycle-components:2.2.1'
```

### 2、配置Activity/Fragment容器

Activity/Fragment需继承RxAppCompatActivity/RxFragment，目前支持的有如下：



## RxLifecycle支持的Component

代码如下：

```
//只需要继承即可public class MainActivity extends RxAppCompatActivity { }
```

### 3、使用compose操作符绑定容器生命周期

有两种方式：

#### 3.1 使用bindToLifecycle()

以Activity为例，在Activity中使用bindToLifecycle()方法，完成Observable发布的事件和当前的组件绑定，实现生命周期同步。从而实现当前组件生命周期结束时，自动取消对Observable订阅，代码如下：

```
public class MainActivity extends RxAppCompatActivity {
 @Override protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main); // 当执行
 onDestory()时，自动解除订阅
 Observable.interval(1, TimeUnit.SECONDS)
 .doOnDispose(new Action() {
 @Override public void run() throws Exception {
 Log.i(TAG, "Unsubscribing subscription from onCreate()");
 }
 })
 .compose(this)
 .subscribe(new Consumer<Long>() {
 @Override public void accept(Long num) throws Exception {
 Log.i(TAG, "Started in onCreate(), running until
onDestory(): " + num);
 }
 });
 }
}
```

#### 3.2 使用bindUntilEvent()

使用ActivityEvent类，其中的CREATE、START、RESUME、PAUSE、STOP、DESTROY分别对应生命周期内的方法。使用bindUntilEvent指定在哪个生命周期方法调用时取消订阅：

```
public class MainActivity extends RxAppCompatActivity {
 @Override protected void onResume() {
 super.onResume(); Observable.interval(1,
 TimeUnit.SECONDS).doOnDispose(new Action() {
 @Override public void run()
 throws Exception { Log.i(TAG,
 "Unsubscribing subscription from onResume()");
 } }) //bindUntilEvent(), 内部
 传入指定生命周期参数 .compose(this.
 <Long>bindUntilEvent(ActivityEvent.DESTROY))
 .subscribe(new Consumer<Long>() {
 @Override public void accept(Long num)
 throws Exception { Log.i(TAG, "Started
 in onResume(), running until in onDestroy(): " + num);
 } }); }}}
```

以上，仅仅需要三步：依赖、继承、compose操作符，即可完成在容器的指定生命周期内，RxJava的自动取消订阅。

## 原理分析

RxLifecycle的原理可以说非常简单。

我们直接看一下这行代码的内部原理：

```
| Observable.compose(this.bindToLifecycle())
```

### 1、RxAppCompatActivity

```
public abstract class RxAppCompatActivity extends
AppCompatActivity implements
LifecycleProvider<ActivityEvent> { //1.实际上
RxAppCompatActivity内部存储了一个BehaviorSubject private
final BehaviorSubject<ActivityEvent> lifecycleSubject =
BehaviorSubject.create(); public final
Observable<ActivityEvent> lifecycle() { return
lifecycleSubject.hide(); } public final <T>
LifecycleTransformer<T> bindUntilEvent(@NonNull
ActivityEvent event) { return
RxLifecycle.bindUntilEvent(lifecycleSubject, event); }
//2.实际上返回了一个LifecycleTransformer public
final <T> LifecycleTransformer<T> bindToLifecycle() {
return
RxLifecycleAndroid.bindActivity(lifecycleSubject); }
//3.Activity不同的生命周期，BehaviorSubject对象会发射对应的
ActivityEvent @Override @CallSuper protected
void onCreate(@Nullable Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
lifecycleSubject.onNext(ActivityEvent.CREATE); }
@Override @CallSuper protected void onStart() {
super.onStart();
lifecycleSubject.onNext(ActivityEvent.START); }
@Override @CallSuper protected void onResume() {
super.onResume();
lifecycleSubject.onNext(ActivityEvent.RESUME); }
@Override @CallSuper protected void onPause() {
lifecycleSubject.onNext(ActivityEvent.PAUSE);
super.onPause(); } @Override @CallSuper
protected void onStop() {
lifecycleSubject.onNext(ActivityEvent.STOP);
super.onStop(); } @Override @CallSuper
protected void onDestroy() {
lifecycleSubject.onNext(ActivityEvent.DESTROY);
super.onDestroy(); }}
```

我们继承的RxAppCompatActivity，其内部实际上存储了一个**BehaviorSubject**，关于**BehaviorSubject**，实际上也还是一个Observable，不了解的朋友可以阅读笔者的前一篇文章，本文不再赘述：

## 理解RxJava（四）Subject用法及原理分析

这个**BehaviorSubject**会在不同的生命周期发射不同的ActivityEvent，比如在onCreate()生命周期发射ActivityEvent.CREATE，在onStop()发射ActivityEvent.STOP。

在2中，我们可以看到，bindToLifecycle()方法实际返回了一个LifecycleTransformer，那么这个LifecycleTransformer是什么呢？

## 2、LifecycleTransformer

```
public final class LifecycleTransformer<T> implements
ObservableTransformer<T, T>,
FlowableTransformer<T, T>,

SingleTransformer<T, T>,
MaybeTransformer<T, T>,

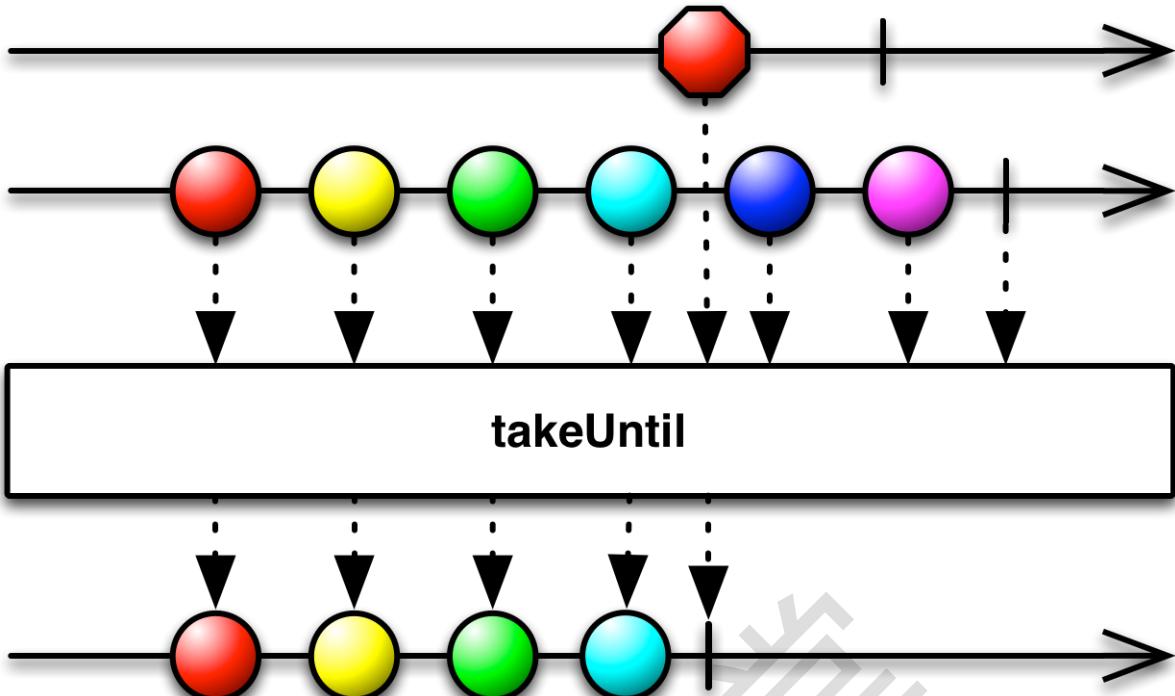
CompletableTransformer{ final Observable<?>
observable; LifecycleTransformer(Observable<?>
observable) { checkNotNull(observable, "observable
== null"); this.observable = observable; }
@Override public ObservableSource<T>
apply(Observable<T> upstream) { return
upstream.takeUntil(observable); } //隐藏其余代码，这里
只以Observable为例}
```

熟悉Transformer的同学都应该能看懂，LifecycleTransformer实际是实现了不同响应式数据类（Observable、Flowable等）的Transformer接口。

以Observable为例，实际上就是通过传入一个Observable序列并存储为成员，然后作为参数给上游的数据源执行takeUntil方法。

### takeUntil操作符

takeUntil操作符:当第二个Observable发射了一项数据或者终止时，丢弃原Observable发射的任何数据。



takeUntil操作符

## 回顾

```
observable.interval(1, TimeUnit.SECONDS)
 .doOnDispose(new Action() { @Override
 public void run() throws Exception {
 Log.i(TAG, "Unsubscribing subscription from
onCreate()); } })
 .compose(this.<Long>bindToLifecycle())
 .subscribe(new Consumer<Long>() {
 @Override public void accept(Long num)
 throws Exception { Log.i(TAG, "Started
in onCreate(), running until onDestory(): " + num);
 } });
}
```

现在回头来看这段代码，就很好理解了，Activity.bindToLifecycle()实际上就是指定上游的数据源，当接收到某个Observable（就是LifecycleTransformer中那个神秘的成员变量）的某个事件时，该数据源自动解除订阅。

老师，原理已经搞清楚了，我还有最后一个问题：

## 神秘人是谁？

回到RxAppCompatActivity中来，我们来看bindToLifecycle()方法：

```
public abstract class RxAppCompatActivity extends
AppCompatActivity implements
LifecycleProvider<ActivityEvent> { private final
BehaviorSubject<ActivityEvent> lifecycleSubject =
BehaviorSubject.create(); public final <T>
LifecycleTransformer<T> bindToLifecycle() { //执行
了这行代码，返回了LifecycleTransformer return
RxLifecycleAndroid.bindActivity(lifecycleSubject); }}
```

不难猜测，实际上，那个神秘人，就是我们RxAppCompatActivity 中的BehaviorSubject成员变量（它本身就是一个Observable）！

我们点进去看看：

```
//1.执行了 bind(lifecycle, ACTIVITY_LIFECYCLE);
public static <T> LifecycleTransformer<T>
bindActivity(@NonNull final Observable<ActivityEvent>
lifecycle) { return bind(lifecycle,
ACTIVITY_LIFECYCLE); } //2.执行了
bind(takeUntilCorrespondingEvent(lifecycle.share(),
correspondingEvents)) public static <T, R>
LifecycleTransformer<T> bind(@NonNull Observable<R>
lifecycle,
 @NonNull final Function<R, R> correspondingEvents)
{ return
bind(takeUntilCorrespondingEvent(lifecycle.share(),
correspondingEvents)); }
```

```
//3.最终抵达这里，这个方法执行了什么呢? private static
<R> Observable<Boolean> takeUntilCorrespondingEvent(final
Observable<R> lifecycle,
 final Function<R,
R> correspondingEvents) { return
Observable.combineLatest(
lifecycle.take(1).map(correspondingEvents),
lifecycle.skip(1), new BiFunction<R, R,
Boolean>() { @Override
public Boolean apply(R bindUntilEvent, R lifecycleEvent)
throws Exception { return
lifecycleEvent.equals(bindUntilEvent); }
}
.onErrorReturn(Functions.RESUME_FUNCTION)
.filter(Functions.SHOULD_COMPLETE); }
```

最后我们走到了3，我们一行一行分析：

### Observable.combineLatest

这行代码实际上是将lifecycle (就是我们传进来的BehaviorSubject) 的事件进行了一次分割：

lifecycle.take(1)指的是最近发射的事件，比如说我们在onCreate()中执行了bindToLifecycle，那么lifecycle.take(1)指的就是ActivityEvent.CREATE，经过map(correspondingEvents)，这个map中传的函数就是1中的ACTIVITY\_LIFECYCLE：

```
private static final Function<ActivityEvent,
ActivityEvent> ACTIVITY_LIFECYCLE = new
Function<ActivityEvent, ActivityEvent>() {
 @Override public ActivityEvent
 apply(ActivityEvent lastEvent) throws Exception {
 switch (lastEvent) {
 case CREATE: return ActivityEvent.DESTROY;
 case START: return ActivityEvent.STOP;
 case RESUME: return ActivityEvent.PAUSE;
 case PAUSE: return ActivityEvent.STOP;
 case STOP: return ActivityEvent.DESTROY;
 case DESTROY: throw new
 OutsideLifecycleException("Cannot bind to Activity
 lifecycle when outside of it.");
 default: throw new
 UnsupportedOperationException("Binding to " + lastEvent +
 " not yet implemented");
 }
 };
```

也就是说，lifecycle.take(1).map(correspondingEvents)实际上是返回了**CREATE** 对应的事件 **DESTROY**，它意味着本次订阅将在Activity的onDestory进行取消。

lifecycle.skip(1)就简单了，除去第一个保留剩下的，以ActivityEvent.Create为例，这里就剩下：

```
ActivityEvent.START
ActivityEvent.RESUME
ActivityEvent.PAUSE
ActivityEvent.STOP
ActivityEvent.DESTROY
```

第三个参数 意味着，lifecycle.take(1).map(correspondingEvents)的序列和lifecycle.skip(1)进行combine，形成一个新的序列：

false,false,fasle,false,true

这意味着，当Activity走到onStart生命周期时，为false,这次订阅不会取消，直到onDestroy，为true，订阅取消。

而后的onErrorReturn和filter是对异常的处理和判断是否应该结束订阅：

```
//异常处理 static final Function<Throwable, Boolean>
RESUME_FUNCTION = new Function<Throwable, Boolean>() {
 @Override public Boolean apply(Throwable
throwable) throws Exception { if (throwable
instanceof OutsideLifecycleException) {
return true; }
Exceptions.propagate(throwable); return false;
} }; //是否应该取消订阅，可以看到，这依赖于上游的
boolean static final Predicate<Boolean>
SHOULD_COMPLETE = new Predicate<Boolean>() {
@Override public boolean test(Boolean
shouldComplete) throws Exception { return
shouldComplete; } };
```

## bind生成LifecycleTransformer

看懂了3，我们回到2，我们生成了一个Observable，然后通过bind(Observable)方法，生成LifecycleTransformer并返回：

```
public static <T, R> LifecycleTransformer<T>
bind(@Nonnull final Observable<R> lifecycle) {
return new LifecycleTransformer<>(lifecycle); }
```

神秘人的神秘面纱就此揭开。

## 总结

RxLifecycle并不难以理解，相反，它的设计思路很简单：

- 1.在Activity中，定义一个Observable (Subject)，在不同的生命周期发射不同的事件；
- 2.通过compose操作符（内部实际上还是依赖takeUntil操作符），定义了上游数据，当其接收到Subject的特定事件时，取消订阅；

3. Subject的特定事件并非是ActivityEvent，而是简单的boolean，它已经内部通过combineLast操作符进行了对应的转化。

实际上，Subject和ActivityEvent对RxLifecycle的使用者来讲，是对应隐藏的。我们只需要调用它提供给我们的API，而内部的实现者我们无需考虑，但是也只有去阅读和理解了它的思想，我们才能更好的选择使用这个库。

## 转折：AutoDispose

在我沉迷于RxLifecycle对项目的便利时，一个机会，我有幸阅读到了[Daniel Lew](#) 的文章[《Why Not RxLifecycle?》（为什么放弃使用RxLifecycle）](#)。

作为RxLifecycle的作者，Daniel Lew客观陈述了使用RxLifecycle在开发时所遇到的窘境。并且为我们提供了一个他认为更优秀的设计：

[AutoDispose: Automatic binding+disposal of RxJava 2 streams.](#)

我花了一些时间研究了一下AutoDispose，不得不承认，AutoDispose相比较RxLifecycle，前者更加健壮，并且拥有更优秀的拓展性，如果我的项目需要一个处理RxJava自动取消订阅的库，也许AutoDispose更为适合。

更让我感到惊喜的是，AutoDispose的设计思想，有很大一部分和RxLifecycle相似，而且我在其文档上，看到了Uber的工程师对于[Daniel Lew](#) 所维护的[RxLifecycle](#) 感谢，也感谢Daniel Lew 对于 AutoDispose 的很大贡献：

以下摘录于AutoDispose的官方文档：

Special thanks go to **Dan Lew** (creator of RxLifecycle), who helped pioneer this area for RxJava in android and humored many of the discussions around lifecycle handling over the past couple years that we've learned from. Much of the internal scope resolution mechanics of

**AutoDispose** are inspired by **RxLifecycle**.

在我尝试将AutoDispose应用在项目中时，我发现国内对于这个库所应用的不多，我更希望有更多朋友能够和我一起使用这个库，于是我准备写一篇关于**AutoDispose**使用方式的文章。

但在这之前，我还是希望能够先将**RxLifecycle**的个人心得分享给大家，原因有二：

一是**RxLifecycle**更简单，原理也更好理解，学习一个库不仅仅是为了会使用它，我们更希望能够从源码中学习到作者优秀的设计和思想。因此，如果是从未接触过这两个库的开发者，直接入手**AutoDispose**，在我看来不如先入手**RxLifecycle**。

二是**AutoDispose**的很大一部分设计核心源于**RxLifecycle**，理解了**RxLifecycle**，更有利于**AutoDispose**的学习。

接下来我将会尝试向大家阐述**AutoDispose**的使用及其内部原理，和**RxLifecycle**不同，我们不再需要去继承**RxActivity/RxFragment**，而是内部依赖使用了google不久前推出的架构组件 **Lifecycle**（事实上这也是我青睐**AutoDispose**的原因之一，尽量避免继承可以给代码更多选择的余地）。

### 20.3.3 RxLifecycle详细解析

#### 一、介绍

RxLifecycle目的：解决RxJava使用中的内存泄漏问题。

例如，当使用RxJava订阅并执行耗时任务后，当Activity被finish时，如果耗时任务还未完成，没有及时取消订阅，就会导致Activity无法被回收，从而引发内存泄漏。

为了解决这个问题，就产生了RxLifecycle，让RxJava变得有生命周期感知，使得其能及时取消订阅，避免出现内存泄漏问题。

#### 二、使用

首先来介绍下RxLifecycle的使用。

##### 1.添加依赖

```
implementation
'com.trello.rxlifecycle2:rxlifecycle:2.2.1'
implementation 'com.trello.rxlifecycle2:rxlifecycle-
android:2.2.1' implementation
'com.trello.rxlifecycle2:rxlifecycle-components:2.2.1'
```

## 2. 继承容器类

Activity/Fragment 需要继承 RxAppCompatActivity/RxFragment，主要支持如下几种容器类：

-  RxAppCompatActivity
-  RxAppCompatDialogFragment
-  RxDialogFragment
-  RxFragment
-  RxFragmentActivity

只需要在项目中针对 base 类的容器中继承实现对应的 Rx 类即可，这一步主要是对生命周期的回调事件进行监听。

## 3. 绑定容器生命周期

以 Activity 为例，主要有如下两种方法：

```
bindUntilEvent(@NonNull ActivityEvent event)
```

```
bindToLifecycle()
```

针对 Fragment 也有同样的两种方法，只是方法名会有所不同。

下面详细介绍这两种方法的区别：

`bindUntilEvent`

该方法指定在哪个生命周期方法调用时取消订阅。

其中 `ActivityEvent` 是一个枚举类，对应于 `Activity` 的生命周期。

```
public enum ActivityEvent { CREATE, START,
 RESUME, PAUSE, STOP, DESTROY}
```

具体使用示例：

```
override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)
 Observable.interval(1, TimeUnit.SECONDS)
 .doOnDispose { Log.i(TAG,
 "Unsubscribing subscription from onDestroy()")
 }
 .compose(bindUntilEvent(ActivityEvent.DESTROY))
 .subscribe { Log.i(TAG, "Started
 in onCreate(), running until in onDestroy(): $it")
 }
}
```

指定在生命周期 `onDestroy()` 时，取消订阅。

`bindToLifecycle`

在某个生命周期进行绑定，在对应的生命周期进行订阅解除。

具体使用示例：

```
override fun onResume() { super.onResume()
 Observable.interval(1, TimeUnit.SECONDS)
 .doOnDispose { Log.i(TAG,
 "Unsubscribing subscription from onPause()")
 }
 .compose(bindToLifecycle())
 .subscribe { Log.i(TAG, "Started
 in onResume(), running until in onPause(): $it")
 }
}
```

在 `onResume()` 进行绑定订阅，则在 `onPause()` 进行解除订阅，生命周期是两两对应的。

### 三、原理解析

1. `compose`

首先来了解一下 `compose` 操作符。

```
compose(bindToLifecycle())compose(bindUntilEvent(ActivityEvent.DESTROY))
```

如上所示，两种绑定生命周期的方式，都是通过 `compose` 操作符进行实现的。

`compose` 一般情况下可以配合 `Transformer` 使用，以实现将一种类型的 `Observable` 转换成另一种类型的 `Observable`，保证调用的链式结构。

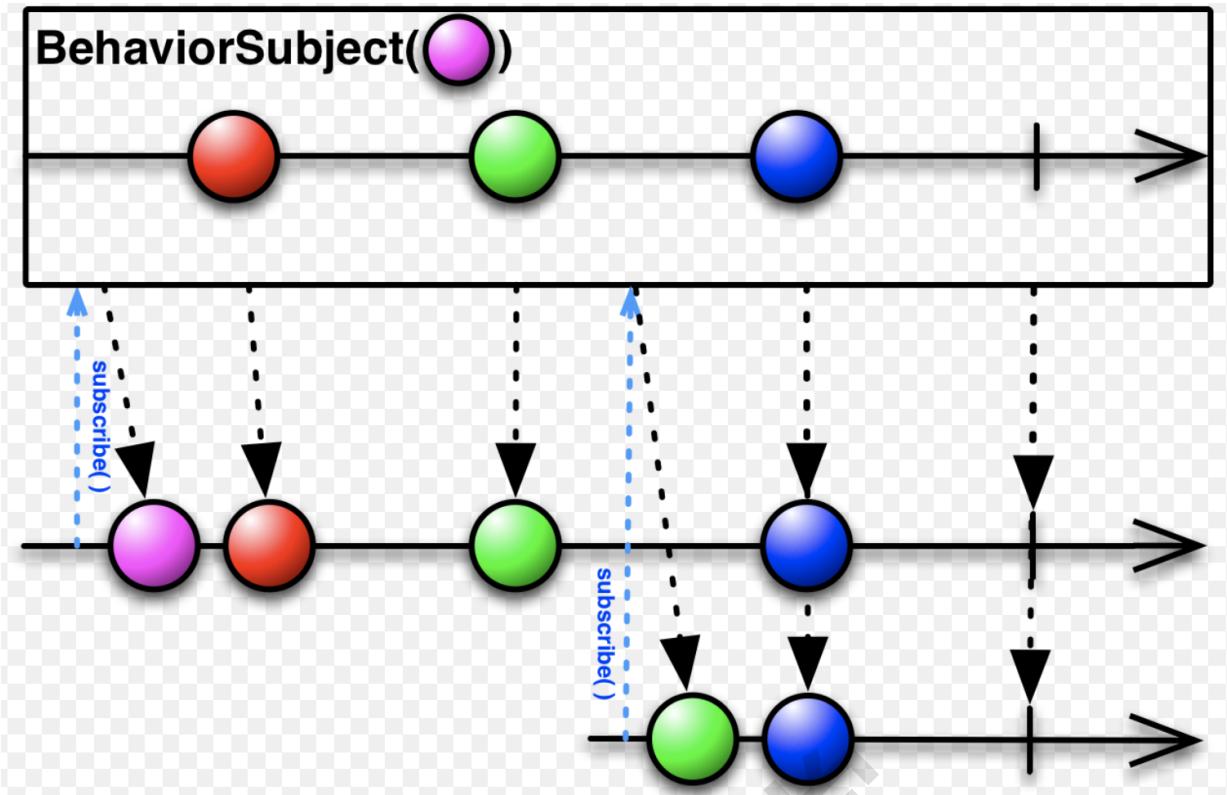
那么接下来看该操作符在 `RxLifecycle` 中的应用，从 `bindToLifecycle` 和 `bindUntilEvent` 入手。

## 2. BehaviorSubject

```
public abstract class RxAppCompatActivity extends
AppCompatActivity implements
LifecycleProvider<ActivityEvent> { private final
BehaviorSubject<ActivityEvent> lifecycleSubject =
BehaviorSubject.create(); @Override @NonNull
@CheckResult public final Observable<ActivityEvent>
lifecycle() { return lifecycleSubject.hide(); }
 @Override @NonNull @CheckResult public final
<T> LifecycleTransformer<T> bindUntilEvent(@NonNull
ActivityEvent event) { return
RxLifecycle.bindUntilEvent(lifecycleSubject, event); }
 @Override @NonNull @CheckResult public final
<T> LifecycleTransformer<T> bindToLifecycle() {
return RxLifecycleAndroid.bindActivity(lifecycleSubject);
} @Override @CallSuper protected void
onCreate(@Nullable Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
lifecycleSubject.onNext(ActivityEvent.CREATE); }
 @Override @CallSuper protected void onStart() {
super.onStart();
lifecycleSubject.onNext(ActivityEvent.START); }
 @Override @CallSuper protected void onResume() {
super.onResume();
lifecycleSubject.onNext(ActivityEvent.RESUME); }
 @Override @CallSuper protected void onPause() {
lifecycleSubject.onNext(ActivityEvent.PAUSE);
super.onPause(); } @Override @CallSuper
protected void onStop() {
lifecycleSubject.onNext(ActivityEvent.STOP);
super.onStop(); } @Override @CallSuper
protected void onDestroy() {
lifecycleSubject.onNext(ActivityEvent.DESTROY);
super.onDestroy(); }}
```

RxAppCompatActivity 中有一个关键对象 BehaviorSubject

BehaviorSubject 会发送离订阅最近的上一个值，没有上一个值的时候会发送默认值。如下图：



所以 `lifecycleSubject` 会根据绑定订阅的时期，不断发送接下来的生命周期事件 `ActivityEvent`。

### 3. `LifecycleTransformer`

接下来继续看源码，`bindToLifecycle` 和 `bindUntilEvent` 都返回了一个 `LifecycleTransformer` 对象，那么 `LifecycleTransformer` 到底有什么用？

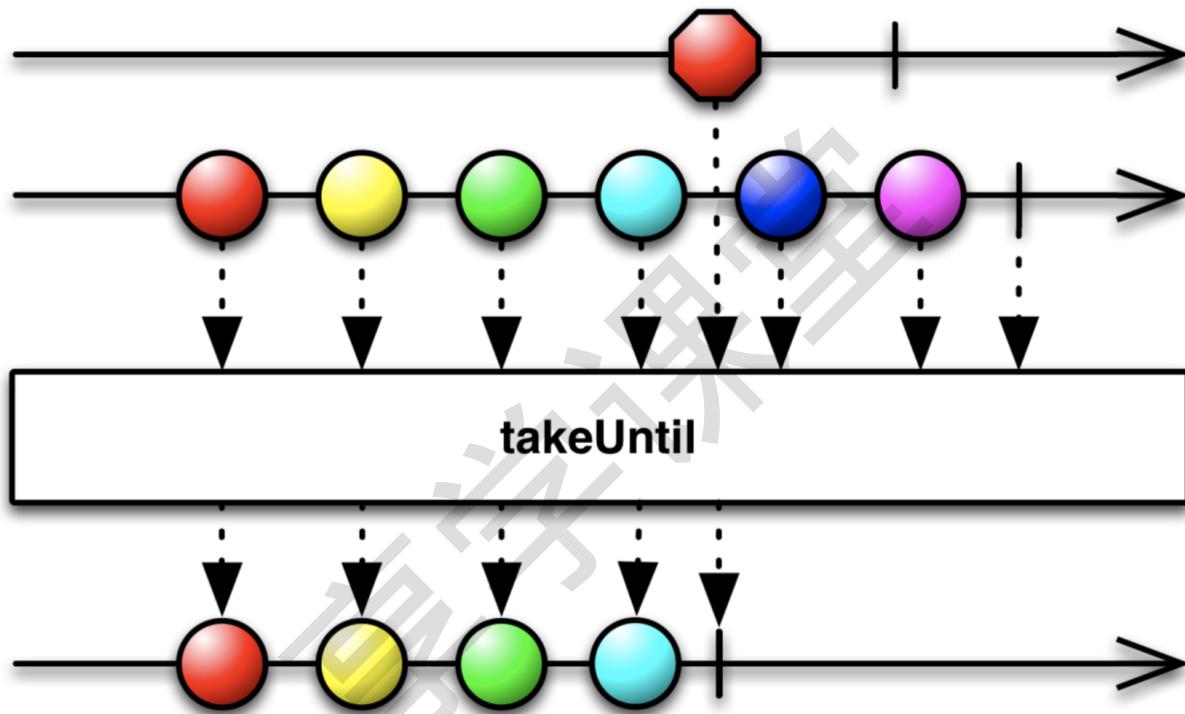
```
@ParametersAreNonnullByDefault
public final class LifecycleTransformer<T> implements ObservableTransformer<T, T>, FlowableTransformer<T, T>, SingleTransformer<T, T>, MaybeTransformer<T, T>, CompletableTransformer {
 final Observable<?> observable;
 LifecycleTransformer(Observable<?> observable) {
 checkNotNull(observable, "observable == null");
 this.observable = observable;
 }
 @Override public ObservableSource<T> apply(Observable<T> upstream) {
 return upstream.takeUntil(observable);
 }
 @Override public Publisher<T> apply(Flowable<T> upstream) {
 return upstream.takeUntil(observable.toFlowable(BackpressureStrategy.LATEST));
 }
 @Override public SingleSource<T> apply(Single<T> upstream) {
 return upstream.takeUntil(observable.firstOrError());
 }
 @Override public MaybeSource<T> apply(Maybe<T> upstream) {
 return upstream.takeUntil(observable.firstElement());
 }
 @Override public CompletableSource apply(Completable upstream) {
 return Completable.ambArray(upstream, observable.flatMapCompletable(Functions.CANCEL_COMPLETED));
 }
 @Override public boolean equals(Object o) {
 if (this == o) { return true; }
 if (o == null || getClass() != o.getClass()) { return false; }
 LifecycleTransformer<?> that = (LifecycleTransformer<?>) o;
 return observable.equals(that.observable);
 }
 @Override public int hashCode() {
 return observable.hashCode();
 }
 @Override public String toString() {
 return "LifecycleTransformer{" +
 "observable=" + observable +
 '}';
 }
}
```

`LifecycleTransformer`实现了各种`Transformer`接口，能够将一个`Observable/Flowable/single/Completable/Maybe`对象转换成另一个`Observable/Flowable/single/Completable/Maybe`对象。正好配合上文的`compose`操作符，使用在链式调用中。

#### 4. `takeUntil`

接下来到了关键了，`LifecycleTransformer`到底把原来的`Observable`对象转换成了什么样子？

这就需要了解`takeUntil`操作符了！



当第二个`Observable`发射了一项数据或者终止时，丢弃原`Observable`发射的任何数据。所谓的第二个`Observable`，即传入`takeUntil`中的`Observable`对象。

理解了该操作符的作用，那么你可能就明白了，`RxLifecycle`就是通过监听第二个`Observable`发射的数据，来解除订阅。

那么这第二个`Observable`是谁？

不就是在创建`LifecycleTransformer`的时候传入构造函数中的嘛，那就来寻找一下什么时候创建的该对象即可。

从头开始捋一捋：

```
public final <T> LifecycleTransformer<T>
bindUntilEvent(@NonNull ActivityEvent event) {
 return RxLifecycle.bindUntilEvent(lifecycleSubject,
 event); }
```

该方法返回了LifecycleTransformer对象，继续向下追溯。

```
public static <T, R> LifecycleTransformer<T>
bindUntilEvent(@Nonnull final Observable<R> lifecycle,
 @Nonnull final R event) { checkNotNull(lifecycle,
 "lifecycle == null"); checkNotNull(event, "event == null");
 return bind(takeUntilEvent(lifecycle,
 event));}private static <R> Observable<R>
takeUntilEvent(final Observable<R> lifecycle, final R
event) { return lifecycle.filter(new Predicate<R>() {
 @Override public boolean test(R
lifecycleEvent) throws Exception { return
lifecycleEvent.equals(event); }});}
```

继续追踪，马上接近真相。

```
public static <T, R> LifecycleTransformer<T>
bind(@Nonnull final Observable<R> lifecycle) { return
new LifecycleTransformer<>(lifecycle);}
```

在该方法中创建了该对象，并传入了一个observable对象，通过上面方法即可知道该对象就是BehaviorSubject对象。

那么该对象在什么时候发送第一次数据呢？

这就要看上面的takeUntilEvent方法了。

关键在这一句lifecycleEvent.equals(event)，只有当BehaviorSubject发送的ActivityEvent的值等于解除绑定的生命周期时，才会发送第一次数据。那么当发送第一次数据时，根据上面的分析就会解除订阅的绑定。

那么针对`bindToLifecycle`方法，是进行怎样的操作，使得在对应的生命周期进行解除订阅呢？

还是继续看源码。

```
public final <T> LifecycleTransformer<T>
bindToLifecycle() { return
RxLifecycleAndroid.bindActivity(lifecycleSubject); }
```

```
public static <T> LifecycleTransformer<T>
bindActivity(@NonNull final Observable<ActivityEvent>
lifecycle) { return bind(lifecycle,
ACTIVITY_LIFECYCLE); }
```

其中`ACTIVITY_LIFECYCLE`为：

```
private static final Function<ActivityEvent,
ActivityEvent> ACTIVITY_LIFECYCLE = new
Function<ActivityEvent, ActivityEvent>() {
@Override public ActivityEvent apply(ActivityEvent
lastEvent) throws Exception { switch
(lastEvent) { case CREATE:
return ActivityEvent.DESTROY; case
START: return ActivityEvent.STOP;
case RESUME: return
ActivityEvent.PAUSE; case PAUSE:
return ActivityEvent.STOP; case
STOP: return ActivityEvent.DESTROY;
case DESTROY: throw new
OutsideLifecycleException("Cannot bind to Activity
lifecycle when outside of it."); default:
throw new
UnsupportedOperationException("Binding to " + lastEvent +
" not yet implemented"); } } };
```

该函数的功能是会根据传入的生命周期事件，返回对应的生命周期，如`CREATE`→`DESTROY`。看来通过该函数就可以实现在对应生命周期解绑了。

不过还需要一系列操作符的协助，继续看源码。

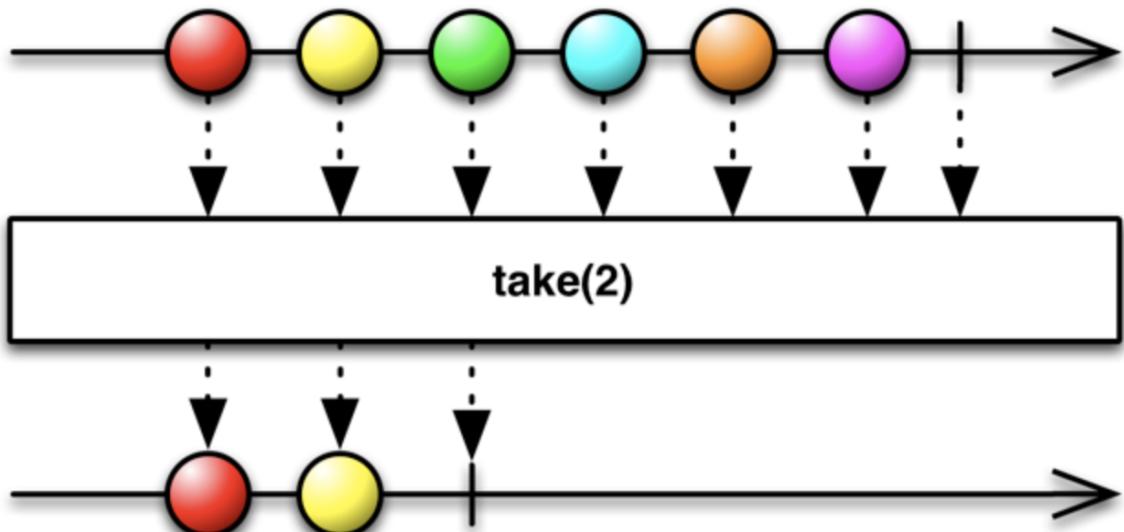
```
public static <T, R> LifecycleTransformer<T>
bind(@NonNull Observable<R> lifecycle,
 @NonNull final
Function<R, R> correspondingEvents) {
checkNotNull(lifecycle, "lifecycle == null");
checkNotNull(correspondingEvents, "correspondingEvents == null");
return
bind(takeUntilCorrespondingEvent(lifecycle.share(),
correspondingEvents)); } private static <R>
Observable<Boolean> takeUntilCorrespondingEvent(final
Observable<R> lifecycle,
 final Function<R,
R> correspondingEvents) {
return
Observable.combineLatest(
lifecycle.take(1).map(correspondingEvents),
lifecycle.skip(1), new BiFunction<R, R,
Boolean>() {
@Override
public Boolean apply(R bindUntilEvent, R lifecycleEvent)
throws Exception {
return
lifecycleEvent.equals(bindUntilEvent); }
})
.onErrorReturn(Functions.RESUME_FUNCTION)
.filter(Functions.SHOULD_COMPLETE); }
```

详细看一下takeUntilCorrespondingEvent方法。

## 5. take

首先看一下take操作符，很简单。

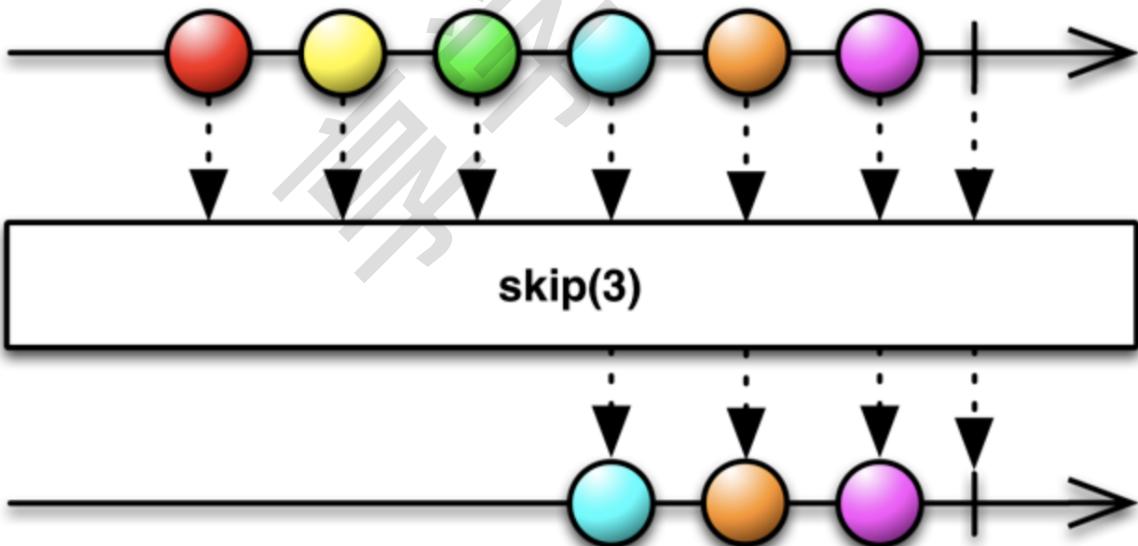
take(int)用一个整数n作为一个参数，只发射前面的n项，如下图：



那么对应 `lifecycle.take(1).map(correspondingEvents)`，即获取发送的第一个生命周期事件，再通过上面对应的函数，转换为响应的生命周期。如果在 `onCreate` 中进行绑定，那么第一个发送的就是 `CREATE`，返回的就是对应的 `DESTORY`。

## 6. `skip`

`skip(int)` 忽略 `Observable` 发射的前  $n$  项数据



`lifecycle.skip(1)`，如果在 `onCreate` 中进行绑定，那么剩余的就是 `START`, `RESUME`, `PAUSE`, `STOP`, `DESTROY`

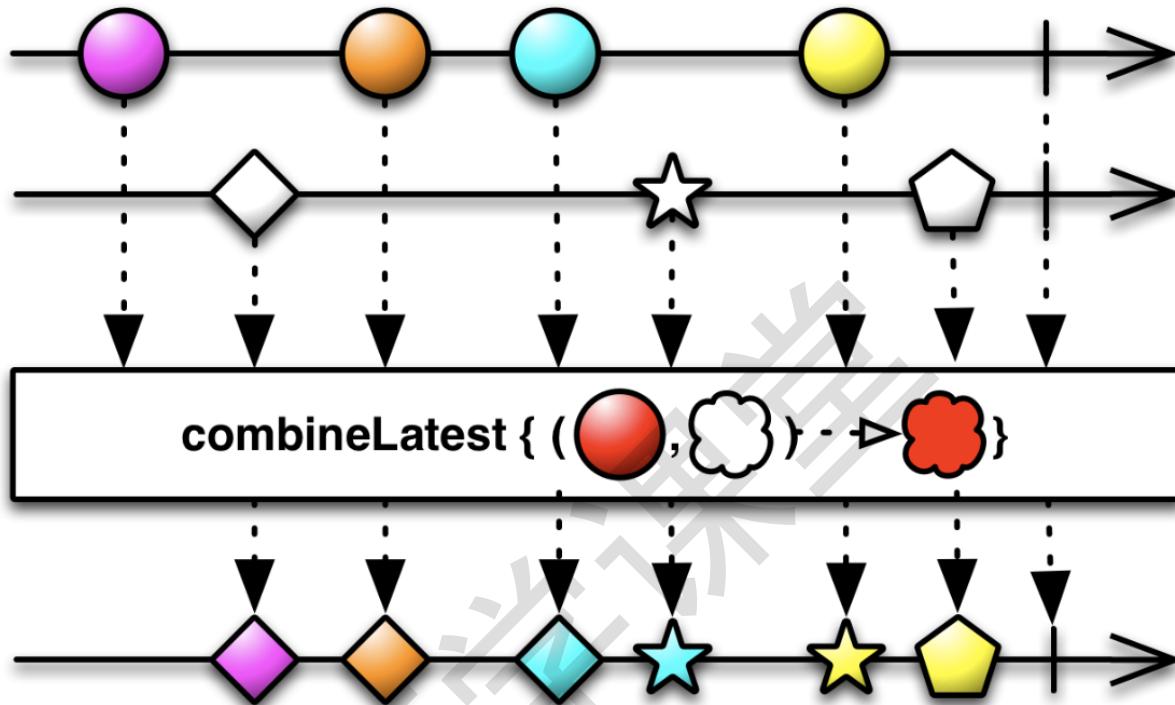
## 7. `combineLatest`

最后还需要一个关键的操作符 `combineLatest`，来完成对应生命周期的解除订阅。

`combineLatest` 操作符可以将2~9个`Observable`发射的数据组装起来然后在发射出来。不过还有两个前提：

- 所有的`Observable`都发射过数据。
- 满足上面条件的时候任何一个`Observable`发射一个数据，就将所有`Observable`最新发射的数据按照提供的函数组装起来发射出去。

具体示例，如下图所示：



按照第三个参数的函数，将  
`lifecycle.take(1).map(correspondingEvents)`和  
`lifecycle.skip(1)`，进行`combine`

```
new BiFunction<R, R, Boolean>() {
 @Override
 public Boolean apply(R bindUntilEvent, R lifecycleEvent) throws Exception {
 return
 }
}
```

那么结果是

```
false, false, false, false, true
```

之后的onErrorReturn和filter是对异常的处理和判断是否应该结束订阅：

```
//异常处理static final Function<Throwable, Boolean>
RESUME_FUNCTION = new Function<Throwable, Boolean>() {
 @Override public Boolean apply(Throwable
throwable) throws Exception { if (throwable
instanceof OutsideLifecycleException) {
return true; } //noinspection
ThrowableResultOfMethodCallIgnored
Exceptions.propagate(throwable); return false;
} }; //是否应该取消订阅，依赖于上游的boolean
static final Predicate<Boolean> SHOULD_COMPLETE = new
Predicate<Boolean>() { @Override public
boolean test(Boolean shouldComplete) throws Exception {
return shouldComplete; } };
```

所以，按照上面的例子，如果在onCreate()方法中进行绑定，那么在onDestory()方法中就会对应的解除订阅。

#### 四、总结

通过上面的分析，可以了解RxLifecycle的使用以及原理。

学习RxLifecycle的过程中，更加体会到了对于观察者模式的使用，以及RxJava操作符的强大，各种操作符帮我们实现一些列的转换。

#### 20.3.4 使用Rxjava2导致的内存泄露问题

Rxjava是个异步库，其链式的api调用使用起来非常简洁，优雅，但是不做处理的话很容易出现内存泄露

##### 内存泄露例子：

有个MainActivity，代码如下：

```
class MainActivity : AppCompatActivity(),
view.OnClickListener { override fun
onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentview(R.layout.activity_main)
btn_start_second.setOnClickListener(this) }
override fun onClick(v: View) { when (v.id) {
R.id.btn_start_second->
{startActivity(Intent(this,SecondActivity::class.java))} }
} }}
```

MainActivity有个按钮，打开SecondActivity，SecondActivity代码如下：

```
class SecondActivity : AppCompatActivity(),
view.OnClickListener { val TAG =
this.javaClass.simpleName override fun
onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentview(R.layout.activity_second) ds =
CompositeDisposable()
btn_start_count.setOnClickListener(this) } override
fun onClick(v: View) { when (v.id) {
R.id.btn_start_count -> {
Observable.interval(1, TimeUnit.SECONDS)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe {
tv_count.text = it.toString()
log(TAG, "count:$it") }
} } } override fun onDestroy() {
super.onDestroy() }}
```

SecondActivity中有个Button，TextView，点击Button，TextView不断输出

**观察测试：**

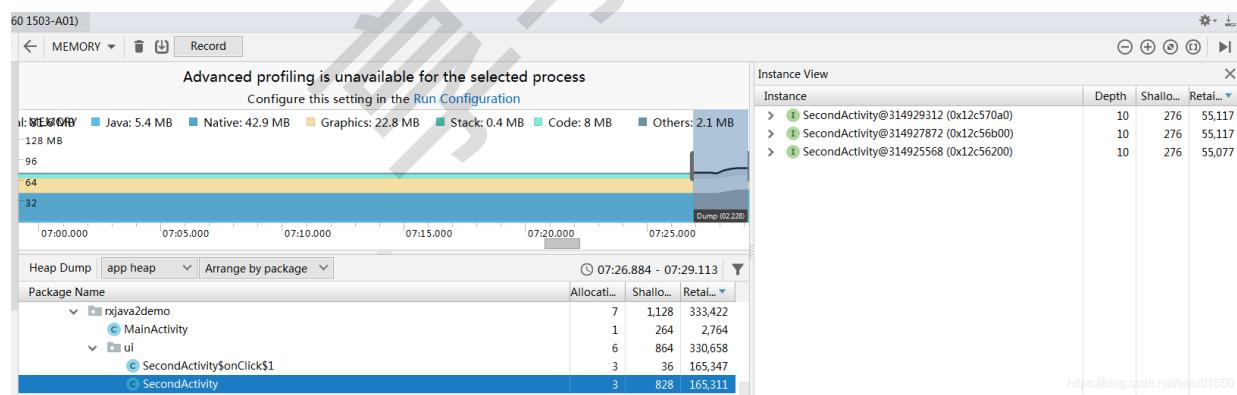
运行App，开启SecondActivity，点击Button改变TextView的值，然后关闭SecondActivity，如此循环操作多次，发现即时关闭了SecondActivity，TextView依旧在输出：

```
01-15 17:29:33.563 26906-26906/com.rain.rxjava2demo
E/SecondActivity: count:14601-15 17:29:33.944 26906-
26906/com.rain.rxjava2demo E/SecondActivity: count:14401-
15 17:29:34.471 26906-26906/com.rain.rxjava2demo
E/SecondActivity: count:15001-15 17:29:34.563 26906-
26906/com.rain.rxjava2demo E/SecondActivity: count:14701-
15 17:29:34.948 26906-26906/com.rain.rxjava2demo
E/SecondActivity: count:145
```

打开Profile分析器，观察Memory情况：



操作几次后，手动调用GC后的内存占用，如下图：



调取当前java堆的快照，发现有多个SecondActivity的引用及SecondActivity#onClick中的泄露

这时SecondActivity的代码做如下更改（只贴关键部分）：

```

private var d: Disposable? = null private lateinit var
ds: CompositeDisposable override fun onclick(v: View) {
 when (v.id) { R.id.btn_start_count -> {
 d = Observable.interval(1, TimeUnit.SECONDS)
 .subscribeOn(Schedulers.io())
 }
 .observeOn(AndroidSchedulers.mainThread())
 .subscribe {
 tv_count.text = it.toString()
 Loge(TAG, "count:$it")
 ds.add(d!!)
 } } } override fun
onDestroy() { super.onDestroy() // 这样处理发
现依然存在内存泄露
 ds.clear() }

```

在onDestroy中，ds.clear(),即所有的disposable调用d.dispose()方法，这样就不会内存泄露了吗？

## 验证测试：

依旧打开SecondActivity，并点击按钮，这样操作3次，先看log：

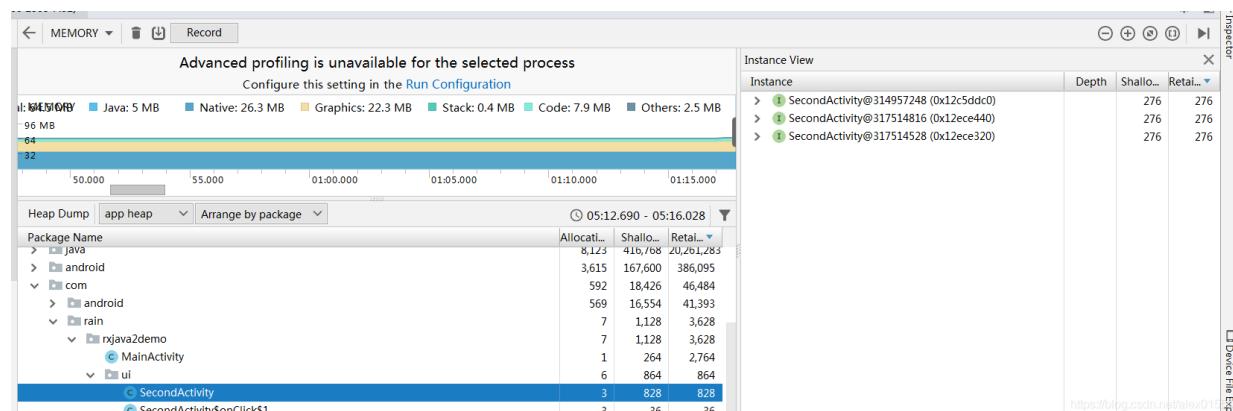
```

01-15 17:49:02.292 7197-7197/com.rain.rxjava2demo
E/SecondActivity: count:001-15 17:49:05.384 7197-
7197/com.rain.rxjava2demo E/SecondActivity: count:001-15
17:49:07.482 7197-7197/com.rain.rxjava2demo
E/SecondActivity: count:0

```

发现subscribe中的代码确实不执行了，

## 看内存堆：



内存占用在65M左右，但是发现依旧存在SecondActivity的引用，先别急，过一段时间再看一次内存占用情况，会发现SecondActivity的引用没有了，虽然手动强制了GC，但是并不一定会立即进行回收，这点要注意。

有文章说d.dispose()只是切除了上下游的数据传递，并没有切断上下游的引用关系，代码做如下修改：

```
d = Observable.interval(1, TimeUnit.SECONDS)
 .onTerminateDetach() // 当执行了d.dispose()方法后
 将解除上下游的引用
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())
 .subscribe {
 tv_count.text = it.toString()
 Loge(TAG, "count:$it")
 }
 ds.add(d!!)
```

代码修改的部分，调用 onTerminateDetach()操作符，看下源码的注释：

```
Nulls out references to the upstream producer and
downstream Observer if the sequence is terminated or
downstream calls dispose().
```

如果序列终止或者下游调用dispose()，那么将解除上下游的引用

~有文章说onTerminateDetach操作符要和subscription.unsubscribe()结合使用，因为不执行subscription.unsubscribe()的话，onTerminateDetach就不会被触发。这时如果我们对当前的stream再进一步操作，比如使用map操作符，那么需要再次调用onTerminateDetach。~

但是我并没有发现二者在内存上的表现有什么不同

**结论：**

1. 在onDestroy方法中调用dispose()方法

**第三方库的使用**

1. 使用[RxLifecycle](#)库，缺点是Activity、Fragment要继承RxActivity、RxFragment

简单使用方法如下：

引入库：

```
implementation
"com.trello.rxlifecycle3:rxlifecycle:$rxlifecycle" implementation
"com.trello.rxlifecycle3:rxlifecycle-
components:$rxlifecycle"
```

Activity、Fragment继承RxActivity、RxFragment等，调用时候绑定对应的生命周期，详细用法请查阅官方文档

```
observable.interval(1, TimeUnit.SECONDS)
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())

 .compose(this.bindUntilEvent(ActivityEvent.DESTROY))
 .subscribe {
 tv_count.text = it.toString()
 Loge(TAG, "count:$it")
 }
 }
```

2. 使用[autodispose](#)库，

引入：

```
implementation "com.uber.autodispose:autodispose-android-
archcomponents-ktx:$autodispose_version" implementation
"com.uber.autodispose:autodispose-lifecycle-
ktx:$autodispose_version"
```

调用如下：

```
observable.interval(1, TimeUnit.SECONDS)
 .subscribeOn(Schedulers.io())
 .observeOn(AndroidSchedulers.mainThread())

 .autoDisposable(AndroidLifecycleScopeProvider.from(this, Lifecycle.Event.ON_DESTROY))
 .subscribe {
 it.toString() tv_count.text =
 "count:$it" loge(TAG,
 } }
```

这里都是演示最基本的用法

## [Demo](#)

### 20.3.5 Rxjava解除订阅②：AutoDispose

#### 前言

上一篇介绍了[Rxjava解除订阅①：自带方式和RxLifeCycle](#)，并且在结尾也透漏了可以解决RxLifeCycle的弊端的方法，那就是AutoDispose。其实AutoDispose大部分也是借鉴RxLifeCycle的经验，很多地方都比较相似，所以源码也比较容易阅读。github地址：[AutoDispose](#)

#### 使用

引入：

androidX以下的版本只支持到0.8.0，android x 并且使用rxjava2，可以使用1.x版本；使用rxjava3的话可以升级到2.x版本。这里以1.3版本为主：

```
def autoDispose = "1.3.0"dependencies {
 implementation "com.uber.autodispose:autodispose:${autoDispose}"
 implementation "com.uber.autodispose:autodispose-android-
 archcomponents:${autoDispose}"}
```

应用起来也很简单：

```
Observable.interval(0, 2, TimeUnit.SECONDS)
 .map(aLong -> {
 Log.d(TAG, "aLong:" + aLong);
 return aLong == 4;
 })
 .as(Dispose.autoDisposable(AndroidLifecycleScopeProvider.from(this, Lifecycle.Event.ON_DESTROY))) //指定解绑生命周期
 .as(Dispose.autoDisposable(AndroidLifecycleScopeProvider.from(this))) // 自动识别生命周期解绑
 .subscribe(aBoolean -> Log.d(TAG, "aBoolean:" + aBoolean));
```

## 使用as操作符，传递

`AutoDispose.autoDisposable(AndroidLifecycleScopeProvider.from(this))`方法即可。跟`RxLifecycle`一样，除了上面这个自动识别生命周期来解绑之外，也提供了`AutoDispose.autoDisposable(AndroidLifecycleScopeProvider.from(this, Lifecycle.Event.ON_DESTROY))`在指定的生命周期解绑操作。

那`as`操作符是什么呢？是将`Observable`转成一个新的对象，跟`compose`不同的地方在于，`compose`是将`Observable`生成的对象还是`Observable`。

## 原理

1.this是什么？

与`RxLifecycle`类似，也是传递当前Activity的`this`进去，不同的是这个`this`并不是Activity本身，而是`LifecycleOwner`。关于`LifecycleOwner`大家自行搜索一下，是google提供的一个生命周期管理的类。

2.生命周期是如何绑定的？

```
as(Dispose.autoDisposable(AndroidLifecycleScopeProvider.from(this)))
```

知道了 `this`，下一步就是看生命周期是如何绑定，进而在合适的生命周期解绑rxjava。`AutoDispose.autoDisposable()` 传递进的是一个 `ScopeProvider`，翻译中文可以认为是一种能力的提供者，那我们传递的 `AndroidLifecycleScopeProvider.from(this)` 返回的就是一个 `AndroidLifecycleScopeProvider`，字面意思就是说 android 生命周期能力提供者。

那么这两方法的作用就知晓了：`AutoDispose.autoDisposable()` 是进行生命周期的解绑方法，而 `AndroidLifecycleScopeProvider.from(this)` 是提供对应的生命周期方法。

`AutoDispose.autoDisposable()` 比较简单：

```
public static <T> AutoDisposeConverter<T>
autoDisposable(final ScopeProvider provider) {
 checkNotNull(provider, "provider == null"); return
 autoDisposable(Completable.defer(() -> {
 try { return
 provider.requestScope(); // 核心方法：最终调用ScopeProvider 的
 requestScope() } catch
 (OutsideScopeException e) { ...
 } }));
}
```

这里用到了 `defer` 操作符，没啥特别的含义，可以简单的理解为懒加载，只有订阅关系成立时才会触发这个方法。

而 `return` 的 `autoDisposable()` 方法则是返回对应的 `ObservableConverter`。这也是因为我们的订阅关系是从 `Observable` 开始的，如果是 `Flowable`，那么对应返回的就是 `FlowableConverter`。而 `autoDisposable()` 内部实现了多个订阅关系的 Converter，无论哪种订阅关系，都可以实现自动解绑。

最终核心的方法是调用了 `provider.requestScope()` 一路追踪源码：

```
public static <E> CompletableSource
resolveScopeFromLifecycle(final
LifecycleScopeProvider<E> provider, final boolean
checkEndBoundary) throws OutsideScopeException {
E lastEvent = provider.peekLifecycle();
CorrespondingEventsFunction<E> eventsFunction =
provider.correspondingEvents(); if (lastEvent == null)
{ throw new LifecycleNotStartedException(); }
E endEvent; try { endEvent =
eventsFunction.apply(lastEvent); // 获取当前生命周期 }
catch (Exception e) { ... } return
resolveScopeFromLifecycle(provider.lifecycle(),
endEvent); }
```

到这，算是知晓了生命周期是从何而来，  
`CorrespondingEventsFunction`会提供给我们最后的一个生命周期方法，无论是自动识别生命周期还是指定生命周期解绑，最终都是通过  
`CorrespondingEventsFunction`得来的。为何要这么说呢？让我们回到  
`from`方法来看：

```
// 在指定生命周期解除绑定public static
AndroidLifecycleScopeProvider from(LifecycleOwner owner,
Lifecycle.Event untilEvent) { return
from(owner.getLifecycle(), untilEvent); } // 自动解除绑定
public static AndroidLifecycleScopeProvider
from(Lifecycle lifecycle) { return from(lifecycle,
DEFAULT_CORRESPONDING_EVENTS); }
```

而自动解除绑定的生命周期判定是`DEFAULT_CORRESPONDING_EVENTS`是这样的：

```
private static final
CorrespondingEventsFunction<Lifecycle.Event>
DEFAULT_CORRESPONDING_EVENTS = lastEvent -> {
switch (lastEvent) { case ON_CREATE:
return Lifecycle.Event.ON_DESTROY; case
ON_START: return Lifecycle.Event.ON_STOP;
case ON_RESUME: return
Lifecycle.Event.ON_PAUSE; case ON_PAUSE:
return Lifecycle.Event.ON_STOP; case ON_STOP:
case ON_DESTROY: default:
throw new LifecycleEndedException("Lifecycle has ended!
Last event was " + lastEvent); } };
```

这个地方可以看到 DEFAULT\_CORRESPONDING\_EVENTS 其实是 CorrespondingEventsFunction 的一个实例，而实现的内容是不是也很眼熟呢？这块跟 RxLifeCycle 一样，自动选择合适生命周期。

再往里走看看指定生命周期的 from 方法是啥：

```
public static AndroidLifecycleScopeProvider from(
Lifecycle lifecycle, Lifecycle.Event untilEvent) {
return from(lifecycle, new
UntilEventFunction(untilEvent)); } // UntilEventFunction 是
CorrespondingEventsFunction 一个具体实现类
private static
class UntilEventFunction implements
CorrespondingEventsFunction<Lifecycle.Event> { private
final Lifecycle.Event untilEvent;
UntilEventFunction(Lifecycle.Event untilEvent) {
this.untilEvent = untilEvent; } @Override public
Lifecycle.Event apply(Lifecycle.Event event) throws
OutsideScopeException { return untilEvent; // 返回指定
的生命周期 } }
```

而 UntilEventFunction 是 CorrespondingEventsFunction 一个具体实现类，最终 apply 方法也是返回了我们指定的那个生命周期。转了一圈到此为止，我们也算是了解了生命周期是如何获取的。那么怎么解绑呢？其实这块内容跟 RxLifeCycle 就非常相似了：

我们继续追踪 `resolveScopeFromLifecycle` 方法到最后实现：

```
public static <E> CompletableSource
resolveScopeFromLifecycle(Observable<E> lifecycle,
final E endEvent, @Nullable final Comparator<E>
comparator) { Predicate<E> equalityPredicate; if
(comparator != null) { equalityPredicate = e ->
comparator.compare(e, endEvent) >= 0; } else {
equalityPredicate = e -> e.equals(endEvent); }
return
lifecycle.skip(1).takeUntil(equalityPredicate).ignoreElements(); }
```

是不是跟 `RxLifeCycle` 的最终核心方法非常相似，也是用的 `takeUntil` 操作符，在指定的生命周期内，打断上层数据流，从而实现自动解绑，避免内存泄漏。

这里简单的说下操作的过程：

- 首先 `skip(1)` 的意思是说，忽略发送的第一个生命周期，因为获取到的第一个生命周期往往都是发送订阅关系的这个生命周期，比如在 `onCreate` 中绑定，那么剩下的生命周期就是 `onStart`, `onResume`, `onPause`, `onDestory` 了。
- 再然后 `takeUntil(equalityPredicate)`，这个操作符比较简单，通过对比判断当前生命周期和指定生命周期是否一致，一致则打断上面的数据流，实现解绑。
- 最终 `ignoreElements` 忽略不执行 `onNext` 方法，只执行 `onComplete` 或者 `onError`。

到此为止，解绑的源码也就分析完了。

## 对比 `RxLifeCycle`

`AutoDispose` 中很多都是借鉴 `RxLifeCycle`，而设计上有些出入：

- 1. `RxLifeCycle` 使用需要实现 `RxAppCompatActivity` 或者 `RxFragment`，对于商业应用来说，随随便便改动基础类是很危险的事情，从设计的角度上来讲：组合的灵活度多数情况下更优于继承。而 `RxLifecycle` 在

父类中声明了一个PublishSubject，用来发射生命周期事件，这是导致其局限性的原因之一。

- 2.在绑定生命周期的设计上，两者也截然不同。`RxLifecycle`需要继承基类，然后从基类中获取生命周期发射到底层，限制了使用场景，对于MVP架构来说，Activity和Fragment作为V层负责ui更新，具体逻辑实现要在P层实现，`RxLifecycle`无法做到在P层自动解绑。而`AutoDispose`是基于google的`LifecycleOwner`实现的，这就大大增加了可扩展性，也就意味着MVP架构的P层实际上也可以直接实现自动解绑操作了。
- 3.同上因为`RxLifecycle`限制使用场景，在我们自定义View中很难应用自动解绑，而`AutoDispose`则可以解决这个问题，并且也提供了对应`viewScopeProvider`，使用跟在Activity中一样，会自动在Detach时解绑。

## MVP架构P层自动解绑

要完成P层自动解绑，需要了解几个类：`LifecycleOwner`，`Lifecycle`，`LifecycleObserver`，具体百度吧。

P层接口继承`LifecycleObserver`：

```
public interface IPresenter extends LifecycleObserver {
 @OnLifecycleEvent(Lifecycle.Event.ON_CREATE) void
 onCreate(LifecycleOwner owner);
 @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY) void
 onDestroy(LifecycleOwner owner);}
```

P层实现：

```
public class BasePresenter implements IPresenter {
 private LifecycleOwner mLifecycleOwner; @Override
 public void onCreate(LifecycleOwner owner) {
 this.mLifecycleOwner = owner; } protected <T>
 AutoDisposeConverter<T> bindLifecycle() { if
 (mLifecycleOwner == null) { throw new
 NullPointerException("未获取LifecycleOwner"); }
 return
 AutoDispose.autoDisposable(AndroidLifecycleScopeProvider.
 from(mLifecycleOwner)); } @Override public void
 onDestroy(LifecycleOwner owner) { }}
```

最后在Activity或者Fragment注册订阅

```
public class Main2Activity extends AppCompatActivity {
 private BasePresenter mPresenter = new BasePresenter();
 @Override protected void onCreate(Bundle
 savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main2);
 getLifecycle().addObserver(mPresenter); }}
```

完成三步，就可以实现P层生命周期监听以及自动解绑操作了。

## 结语

经过对比可看出AutoDispose在设计上更优于RXLifecycle，无论是扩展性还是解耦上都比RXLifecycle要强一些。而对于我们使用者来说，只要能够满足需求，就是好库。

## 20.2 Tinker源码分析

## 20.3 ARouter源码分析

## 路由框架原理：ARouter 【官方文档】

【ARouter：现在模块化（组件化）算是比较常见的开发模式了，尤其是在大厂，所以有必要知道ARouter的实现原理，2个模块之间是怎样实现的页面跳转，以及ARouter为了优化性能做了哪些工作】

核心要点：将Route注解中的path地址和Activity.class文件的映射关系保存到它自己生成的java文件的map中

隐式跳转方案，但是一个项目中不可能所有的跳转都是隐式的，这样Manifest文件会有很多过滤配置，而且非常不利于后期维护。

反射跳转方案：需要拿到Activity的类文件，在组件开发的时候，想拿到其他module的类文件是很麻烦的，因为组件开发的时候组件module之间是没有相互引用的，你只能通过找到类的路径去反射拿到这个class。大量的使用反射跳转对性能会有影响。

APT是Annotation Processing Tool的简称，即注解处理工具。apt是在编译期对代码中指定的注解进行解析，然后做一些其他处理（如通过javapoet生成新的Java文件

### 20.3.1 ARouter原理剖析及手动实现

#### 前言

路由跳转在项目中用了一段时间了，最近对Android中的ARouter路由原理也是研究了一番，于是就给大家分享一下自己的心得体会，并教大家如何实现一款简易的路由框架。

本篇文章分为两个部分，第一部分着重剖析ARouter路由的原理，第二部分会带着大家仿照ARouter撸一个自己的路由框架，我们自己撸的路由框架可能没有Arouter众多的功能如过滤器、provider等，但是却实现了ARouter最核心的功能：路由跳转，同时你也能学会如何去设计一个框架等等。

这里先附上我自己实现的路由框架demo地址：[ARouter原理剖析及手动实现，demo点我访问，欢迎star](#)

第一部分：ARouter原理剖析

说到路由便不得不提一下Android中的组件化开发思想，组件化是最近比较流行的架构设计方案，它能对代码进行高度的解耦、模块分离等，能极大地提高开发效率(如有同学对组件化有不理解，可以参考网上众多的博客等介绍，然后再阅读demo源码中的组件化配置进行熟悉)。路由和组件化本身没有什么联系，因为路由的责任是负责页面跳转，但是组件化中两个单向依赖的module之间需要互相启动对方的Activity，因为没有相互引用，`startActivity()`是实现不了的，必须需要一个协定的通信方式，此时类似ARouter和ActivityRouter等的路由框架就派上用场了。

- 第一节：ARouter路由跳转的原理

如上图，在组件化中，为了业务逻辑的彻底解耦，同时也为了每个module都可以方便的单独运行和调试，上层的各个module不会进行相互依赖(只有在正式联调的时候才会让app壳module去依赖上层的其他组件module)，而是共同依赖于base module，base module中会依赖一些公共的第三方库和其他配置。那么在上层的各个module中，如何进行通信呢？

我们知道，传统的Activity之间通信，通过`startActivity(intent)`，而在组件化的项目中，上层的module没有依赖关系(即便两个module有依赖关系，也只能是单向的依赖)，那么假如login module中的一个Activity需要启动pay\_module中的一个Activity便不能通过`startActivity`来进行跳转。那么大家想一下还有什么其他办法呢？可能有同学会想到隐式跳转，这当然也是一种解决方法，但是一个项目中不可能所有的跳转都是隐式的，这样Manifest文件会有很多过滤配置，而且非常不利于后期维护。当然你用反射拿到Activity的class文件也可以实现跳转，但是第一：大量的使用反射跳转对性能会有影响，第二：你需要拿到Activity的类文件，在组件开发的时候，想拿到其他module的类文件是很麻烦的，因为组件开发的时候组件module之间是没有相互引用的，你只能通过找到类的路径去反射拿到这个class，那么有没有一种更好的解决办法呢？办法当然是有的。下面看图：

在组件化中，我们通常都会在base\_module上层再依赖一个router\_module,而这个router\_module就是负责各个模块之间页面跳转的。

用过ARouter路由框架的同学应该都知道，在每个需要对其他module提供调用的Activity中，都会声明类似下面@Route注解，我们称之为路由地址

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity { @Override protected
void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main); } }
@Route(path = "/module1/module1main")public class Module1MainActivity
extends AppCompatActivity { @Override protected
void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_module1_main); } }
```

那么这个注解有什么用呢，路由框架会在项目的编译期通过注解处理器扫描所有添加@Route注解的Activity类，然后将Route注解中的path地址和Activity.class文件映射关系保存到它自己生成的java文件中。为了让大家理解，我这里来使用近乎伪代码给大家简单演示一下。

```
public class MyRouters{ //项目编译后通过apt生成如下方法
public static HashMap<String, ClassBean>
getRouteInfo(HashMap<String, ClassBean> routes) {
route.put("/main/main", MainActivity.class);
route.put("/module1/module1main",
Module1MainActivity.class);
route.put("/login/login", LoginActivity.class); } }
```

这样我们想在app模块的MainActivity跳转到login模块的LoginActivity，那么便只需调用如下：

```
//不同模块之间启动Activity
public void login(String name,
String password) {
 HashMap<String, ClassBean> route =
MyRouters.getRouteInfo(new HashMap<String, ClassBean>);
LoginActivity.class classBean =
route.get("/login/login"); Intent intent = new
Intent(this, classBean); intent.putExtra("name",
name); intent.putExtra("password", password);
startActivity(intent);}
```

这样是不是很简单就实现了路由的跳转，既没有隐式意图的繁琐，也没有反射对性能的损耗。用过ARouter的同学应该知道，用ARouter启动Activity应该是下面这个写法

```
// 2. Jump with
parametersARouter.getInstance().build("/test/login")
 .withString("password", 666666)
 .withString("name", "小三") .navigation();
```

那么ARouter背后是怎么样实现跳转的呢？实际上它的核心思想跟上面讲解是一样的，我们在代码里加入的@Route注解，会在编译时期通过apt生成一些存储path和activity.class映射关系的类文件，然后app进程启动的时候会加载这些类文件，把保存这些映射关系的数据读到内存里(保存在map里)，然后在进行路由跳转的时候，通过build()方法传入要到达页面的路由地址，ARouter会通过它自己存储的路由表找到路由地址对应的Activity.class(activity.class = map.get(path))，然后new Intent(context, activity.Class)，当调用ARouter的withString()方法它的内部会调用intent.putExtra(String name, String value)，调用navigation()方法，它的内部会调用startActivity(intent)进行跳转，这样便可以实现两个相互没有依赖的module顺利的启动对方的Activity了。

- 第二节：ARouter映射关系如何生成

通过上节我们知道在Activity类上加上@Route注解之后，便可通过apt生成对应的路由表。那么现在我们来搞清楚，既然路由和Activity的映射关系我们可以很容易地得到(因为代码都是我们写的，当然很容易得到)，那么为什么我们要繁琐的通过apt来生成类文件而不是自己直接写一个契约类来保存映射关系呢。如果站在一个框架开发者的角度去理解，就不难明

白了，因为框架是给上层业务开发者调用的，如果业务开发者在开发页面的过程中还要时不时的更新或更改契约类文件，不免过于麻烦？如果有自动根据路由地址生成映射表文件的技术该多好啊！

技术当然是有的，那就是被众多框架使用的apt及javapoet技术，那么什么是apt，什么是javapoet呢？我们先来看下图：

APT是Annotation Processing Tool的简称，即注解处理工具。由图可知，apt是在编译期对代码中指定的注解进行解析，然后做一些其他处理（如通过javapoet生成新的Java文件）。我们常用的ButterKnife，其原理就是通过注解处理器在编译期扫描代码中加入的@BindView、@OnClick等注解进行扫描处理，然后生成XXX\_ViewBinding类，实现了view的绑定。javapoet是鼎鼎大名的squareup出品的一个开源库，是用来生成java文件的一个library，它提供了简便的api供你去生成一个java文件。可以如下引入javapoet

```
implementation 'com.squareup:javapoet:1.7.0'
```

下面我通过demo中的例子带你了解如何通过apt和javapoet技术生成路由映射关系的类文件：

首先第一步，定义注解：

```
@Target(ElementType.TYPE)@Retention(RetentionPolicy.CLASS)
public @interface Route { /** * 路由的路径 *
@return */ String path(); /** * 将路由节点进行
分组，可以实现动态加载 * @return */ String group()
default "";}
```

这里看到Route注解里有path和group，这便是仿照ARouter对路由进行分组。因为当项目变得越来越庞大的时候，为了便于管理和减小首次加载路由表过于耗时的问题，我们对所有的路由进行分组。在ARouter中会要求路由地址至少需要两级，如"/xx/xx"，一个模块下可以有多个分组，这里我们就将路由地址定为必须大于等于两级，其中第一级是group。

第二步，在Activity上使用注解

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity {}@Route(path =
"/main/main2")public class Main2Activity extends
AppCompatActivity {}@Route(path = "/show/info")public
class ShowActivity extends AppCompatActivity {}
```

第三步，编写注解处理器，在编译器找到加入注解的类文件，进行处理，这里我只展示关键代码，具体的细节还需要你去demo中仔细研读：

```
@AutoService(Processor.class)/** 处理器接收的参数 替代
{@link AbstractProcessor#getSupportedOptions()} 函数
*/@SupportedOptions(Constant.ARGUMENTS_NAME)/** * 注册给哪
些注解的 替代 {@link
AbstractProcessor#getSupportedAnnotationTypes()} 函数
*/@SupportedAnnotationTypes(Constant.ANNOTATION_TYPE_ROUTE)
public class RouterProcessor extends AbstractProcessor
{ /** * key:组名 value:类名 */ private
Map<String, String> rootMap = new TreeMap<>(); /**
* 分组 key:组名 value:对应组的路由信息 */ private
Map<String, List<RouteMeta>> groupMap = new HashMap<>();
 /** * @param set 使用了支持处理注解的节点集合
* @param roundEnvironment 表示当前或是之前的运行环境，可以通过该
对象查找找到的注解。 * @return true 表示后续处理器不会再处理
(已经处理) */ @Override public boolean
process(Set<? extends TypeElement> set, RoundEnvironment
roundEnvironment) { if (!utils.isEmpty(set)) {
 //被Route注解的节点集合 Set<? extends
Element> rootElements =
roundEnvironment.getElementsAnnotatedwith(Route.class);
 if (!utils.isEmpty(rootElements)) {
processorRoute(rootElements); }
return true; } return false; } //...}
```

如代码中所示，要想在编译期对注解做处理，就需要RouterProcessor继承自AbstractProcessor并通过@AutoService注解进行注册，然后实现process()方法。还没有完，你还需要通过@SupportedAnnotationTypes(Constant.ANNOTATION\_TYPE\_ROUTE)

指定要处理哪个注解，Constant.ANNOTATION\_TYPE\_ROUTE便是我们的Route注解的路径。看process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment)方法，set集合就是编译期扫描代码得到的加入了Route注解的文件集合，然后我们就可以在process方法生成java文件了。

这里的@AutoService是为了注册注解处理器，需要我们引入一个google开源的自动注册工具AutoService，如下依赖(当然也可以手动进行注册，不过略微麻烦，这里不太推荐)：

```
implementation 'com.google.auto.service:auto-service:1.0-rc2'
```

第四步：通过javapoet生成java类：

在第三步中process()方法里有一句代码：

processorRoute(rootElements)，这个就是生成java文件的方法了，下面我贴出代码：

```
private void processorRoute(Set<? extends Element> rootElements) { //... //生成Group记录分组表
generatedGroup(iRouteGroup); //生成Root类 作用：记录<分组，对应的Group类> generatedRoot(iRouteRoot,
iRouteGroup);}
```

processorRoute()方法内容很多，这里我只贴出生成java文件相关，其他代码我会在第二部分手动实现路由框架中详细介绍。如上，generatedGroup(iRouteGroup)和generatedRoot(iRouteRoot, iRouteGroup)就是生成java文件的核心了。这里我只贴出generatedRoot()方法，因为生成类文件的原理都是一样的，至于生成什么功能的类，只要你会一个，举一反三，这便没有什么难度。

想  
識  
思  
學

```
/** * 生成Root类 作用：记录<分组，对应的Group类> * @param iRouteRoot * @param iRouteGroup */private void generatedRoot(TypeElement iRouteRoot, TypeElement iRouteGroup) { //创建参数类型 Map<String,Class<? extends IRouteGroup>> routes> //Wildcard 通配符 ParameterizedTypeName parameterizedTypeName = ParameterizedTypeName.get(ClassName.get(Map.class), ClassName.get(String.class), ParameterizedTypeName.get(ClassName.get(Class.class), WildcardTypeName.subtypeOf(ClassName.get(iRouteGroup))))； //生成参数 Map<String,Class<? extends IRouteGroup>> routes ParameterSpec parameter = ParameterSpec.builder(parameterizedTypeName, "routes").build(); //生成函数 public void loadInfo(Map<String,Class<? extends IRouteGroup>> routes) MethodSpec.Builder methodBuilder = MethodSpec.methodBuilder(Constant.METHOD_LOAD_INTO) .addModifiers(Modifier.PUBLIC) .addAnnotation(Override.class) .addParameter(parameter); //生成函数体 for (Map.Entry<String, String> entry : rootMap.entrySet()) { methodBuilder.addStatement("routes.put($S, $T.class)", entry.getKey(), ClassName.get(Constant.PACKAGE_OF_GENERATE_FILE, entry.getValue())); } //生成$Root$类 String className = Constant.NAME_OF_ROOT + moduleName; TypeSpec typeSpec = TypeSpec.classBuilder(className) .addSuperinterface(ClassName.get(iRouteRoot)) .addModifiers(Modifier.PUBLIC) .addMethod(methodBuilder.build()); try { //生成java文件， PACKAGE_OF_GENERATE_FILE就是生成文件需要的路径 JavaFile.builder(Constant.PACKAGE_OF_GENERATE_FILE, typeSpec).build().writeTo(filerUtils); log.i("Generated RouteRoot: " + Constant.PACKAGE_OF_GENERATE_FILE + "." + className); } }
```

```
} catch (IOException e) { e.printStackTrace();
}
```

如上，我把每一块代码的作用注释了出来，相信大家很容易就能理解每一个代码段的作用。可见，其实生成文件只是调用一些api而已，只要我们熟知api的调用，生成java文件便没有什么难度。

## 第二部分：动手实现一个路由框架

通过第一部分的讲述，我相信大家对于ARouter的原理已经有了整体轮廓的理解，这一部分，我便会通过代码带你去实现一个自己的路由框架。要实现这个路由框架，我们先来实现生成路由映射文件这一块，因为这一块是路由框架能够运行起来的核心。

- 第一节：生成路由映射文件

通过第一部分的讲述我们知道在Activity类上加上@Route注解之后，便可通过apt来生成对应的路由表，那么现在我们就来生成这些路由映射文件。首先，我们要理解一个问题，就是我们的路由映射文件是在编译期间生成的，那么在程序的运行期间我们要统一调用这些路由信息，便需要一个统一的调用方式。我们先来定义这个调用方式：

```
public interface IRouteGroup { void
loadInto(Map<String, RouteMeta> atlas); } public interface
IRouteRoot { void loadInto(Map<String, Class<? extends
IRouteGroup>> routes); }
```

我们定义两个接口来对生成的java文件进行约束，IRouteGroup是生成的分组关系契约，IRouteRoot是单个分组路由信息契约，只要我们生成的java文件继承自这个接口并实现loadInto()方法，在运行期间我们就可以统一的调用生成的java文件，获取路由映射信息。

现在我们来把RouterProcessor生成路由映射文件相关的代码补全：

想  
識  
思  
學

```
@AutoService(Processor.class)/** 处理器接收的参数 替代
{@link AbstractProcessor#getSupportedOptions()} 函数
*/@SupportedOptions(Constant.ARGUMENTS_NAME)/** * 指定使用的Java版本 替代 {@link
AbstractProcessor#getSupportedSourceversion()} 函数
*/@SupportedSourceversion(SourceVersion.RELEASE_7)/** * 注册给哪些注解的 替代 {@link
AbstractProcessor#getSupportedAnnotationTypes()} 函数
*/@SupportedAnnotationTypes(Constant.ANNOTATION_TYPE_ROUT
E)
public class RouterProcessor extends AbstractProcessor
{
 /** * key:组名 value:类名 */
 private Map<String, String> rootMap = new TreeMap<>();
 /**
 * 分组 key:组名 value:对应组的路由信息 */
 private Map<String, List<RouteMeta>> groupMap = new HashMap<>();
 /**
 * 节点工具类 (类、函数、属性都是节点) */
 private Elements elementutils; /**
 * type(类信息)工具类 */
 private Types typeutils; /**
 * 文件生成器 类/资源 */
 private Filer filerutils; private
String moduleName; private Log log; @Override
public synchronized void init(ProcessingEnvironment
processingEnvironment) {
 super.init(processingEnvironment); //获得apt的日志输出
 log =
Log.newLog(processingEnvironment.getMessager());
 elementutils = processingEnvironment.getElementUtils();
 typeutils = processingEnvironment.getTypeUtils();
 filerutils = processingEnvironment.getFiler();
 //参数是模块名 为了防止多模块/组件化开发的时候 生成相同的
 xx$$ROOT$$文件
 Map<String, String> options =
processingEnvironment.getOptions(); if
(!utils.isEmpty(options)) {
 moduleName =
options.get(Constant.ARGUMENTS_NAME); } if
(utils.isEmpty(moduleName)) {
 throw new
RuntimeException("Not set processor moudleName option
!"); } log.i("init RouterProcessor " +
moduleName + " success !"); } /**
 * @param set 使用了支持处理注解的节点集合
 * @param roundEnvironment 表示当前或是之前的运行环境,可以通过该对象查找找
```

```
到的注解。 * @return true 表示后续处理器不会再处理(已经处理)
 */
@Override public boolean process(Set<?
extends TypeElement> set, RoundEnvironment
roundEnvironment) { if (!utils.isEmpty(set)) {
//被Route注解的节点集合
set<? extends
Element> rootElements =
roundEnvironment.getElementsAnnotatedWith(Route.class);
if (!utils.isEmpty(rootElements)) {
processorRoute(rootElements);
}
return true;
} return false;
} //...
}
```

我们通过@SupportedOptions(Constant.ARGUMENTS\_NAME)拿到每个module的名字，用来生成对应module下存放路由信息的类文件名。这里变量Constant.ARGUMENTS\_NAME的值就是moduleName，在这之前，我们需要在每个组件module的gradle下配置如下

```
javaCompileOptions {
annotationProcessorOptions {
[moduleName: project.getName()]
arguments =
}}
```

@SupportedAnnotationTypes(Constant.ANNOTATION\_TYPE\_ROUTE) 指定了需要处理的注解的路径地址,在此就是Route.class的路径地址。

RouterProcessor中我们实现了init方法，拿到log apt日志输出工具用以输出apt日志信息,并通过以下代码得到上面提到的每个module配置的moduleName

```
//参数是模块名 为了防止多模块/组件化开发的时候 生成相同的
xx$$ROOT$$文件Map<String, String> options =
processingEnvironment.getOptions();if
(!utils.isEmpty(options)) { moduleName =
options.get(Constant.ARGUMENTS_NAME); }if
(utils.isEmpty(moduleName)) { throw new
RuntimeException("Not set processor moudleName option
!"); }
```

然后在process()方法里开始生成文件名以 EaseRouter\_Route\_moduleName和EaseRouter\_Group\_moduleName 命名的文件。(这里的moduleName指具体的module名，demo中apt相关的代码实现都在easy-compiler module中)，生成 EaseRouter\_Route\_moduleName相关文件存储的就是分组关系，生成 EaseRouter\_Group\_moduleName相关文件里存储的就是分组下的路由映射关系。

好了，我们终于可以生成文件了，在process()方法里有如下代码，

```
if (!utils.isEmpty(set)) { //被Route注解的节点集合
Set<? extends Element> rootElements =
roundEnvironment.getElementsAnnotatedwith(Route.class);
if (!utils.isEmpty(rootElements)) {
processorRoute(rootElements); } return true;}return
false;
```

set就是扫描得到的支持处理注解的节点集合，然后得到rootElements，即被@Route注解的节点集合，此时就可以调用 processorRoute(rosoElements)方法去生成文件了。  
processorRoute(rootElements)方法实现如下：

```
private void processorRoute(Set<? extends Element>
rootElements) { //获得Activity这个类的节点信息
TypeElement activity =
elementutils.getTypeElement(Constant.ACTIVITY);
TypeElement service =
elementutils.getTypeElement(Constant.ISERVICE); for
(Element element : rootElements) { RouteMeta
routeMeta; //类信息 TypeMirror typeMirror =
element.asType(); log.i("Route class:" +
typeMirror.toString()); Route route =
element.getAnnotation(Route.class); if
(typeutils.isSubtype(typeMirror, activity.asType())) {
routeMeta = new
RouteMeta(RouteMeta.Type.ACTIVITY, route, element);
} else if (typeutils.isSubtype(typeMirror,
service.asType())) { routeMeta = new
RouteMeta(RouteMeta.Type.ISERVICE, route, element);
} else { throw new RuntimeException("Just
support Activity or IService Route: " + element);
} categories(routeMeta); } TypeElement
iRouteGroup =
elementutils.getTypeElement(Constant.IROUTE_GROUP);
TypeElement iRouteRoot =
elementutils.getTypeElement(Constant.IROUTE_ROOT); //生
成Group记录分组表 generatedGroup(iRouteGroup); //生成
Root类 作用：记录<分组，对应的Group类>
generatedRoot(iRouteRoot, iRouteGroup); }
```

上面提到的生成的EaseRouter\_Route\_moduleName文件和  
EaseRouter\_Group\_moduleName文件分别实现了IRouteRoot和  
IRouteGroup接口，就是通过下面这两行代码拿到IRootGroup和  
IRootRoot的字节码信息，然后传入generatedGroup(iRouteGroup)和  
generatedRoot(iRouteRoot, iRouteGroup)方法，这两个方法内部会通过  
javapoet api生成java文件，并实现这两个接口。

```
TypeElement iRouteGroup =
elementutils.getTypeElement(Constant.IROUTE_GROUP);TypeEl
ement iRouteRoot =
elementutils.getTypeElement(Constant.IROUTE_ROOT);
```

generatedGroup(iRouteGroup)和generatedRoot(iRouteRoot, iRouteGroup)就是生成上面提到的EaseRouter\_Root\_app和 EaseRouter\_Group\_main等文件的具体实现，生成的方法我在第一部分已经贴出来过了，这里不再阐述。

好了，现在我们编译下项目就会在每个组件module的 build/generated/source/apt目录下生成相关映射文件。这里我把app module编译后生成的文件贴出来，app module编译后会生成 EaseRouter\_Root\_app文件和EaseRouter\_Group\_main、 EaseRouter\_Group\_show等文件， EaseRouter\_Root\_app文件对应于 app module的分组，里面记录着本module下所有的分组信息， EaseRouter\_Group\_main、 EaseRouter\_Group\_show文件分别记载着当前分组下的所有路由地址和ActivityClass映射信息。如下所示：

```
public class EaseRouter_Root_app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("main",
 EaseRouter_Group_main.class); routes.put("show",
 EaseRouter_Group_show.class); }
 public class EaseRouter_Group_main implements IRouteGroup {
 @Override public void loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/main/main", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, Main2\Activity.class, "/main/main", "main"));
 atlas.put("/main/main2", RouteMeta.build(RouteMeta.Type.AC
 TIVITY, Main2\Activity.class, "/main/main2", "main"));
 }
 public class EaseRouter_Group_show implements
 IRouteGroup {
 @Override public void
 loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/show/info", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, ShowActivity.class, "/show/info", "show"));
 }
 }
 }
}
```

大家会看到生成的类分别实现了IRouteRoot和IRouteGroup接口，并且实现了loadInto()方法，而loadInto方法通过传入一个特定类型的map就能把分组信息放入map里，只要分组信息存入到特定的map里后，我们就可以随意的从map里取路由地址对应的Activity.class做跳转使用。那么如果我们在login\_module中想启动app\_module中的MainActivity类，首先，我们已知MainActivity类的路由地址是"/main/main"，第一个"/main"代表分组名，那么我们岂不是可以像下面这样调用去得到MainActivity类文件，然后startActivity()跳转到MainActivity。(这里的RouteMeta只是存有Activity class文件的封装类，先不用理会)。

```
public void test() { EaseRouter_Root_app rootApp = new
EaseRouter_Root_app(); HashMap<String, Class<? extends
IRouteGroup>> rootMap = new HashMap<>();
rootApp.loadInto(rootMap); //得到/main分组 Class<?
extends IRouteGroup> aClass = rootMap.get("main"); try
{
 HashMap<String, RouteMeta> groupMap = new
HashMap<>();
aClass.newInstance().loadInto(groupMap); //得到
MainActivity RouteMeta main =
groupMap.get("/main/main"); Class<?>
mainActivityClass = main.getDestination(); Intent
intent = new Intent(this, mainActivityClass);
startActivity(intent); } catch (InstantiationException
e) { e.printStackTrace(); } catch
(IllegalAccessException e) { e.printStackTrace();
}}
```

可以看到，只要有了这些附带路由映射信息的类文件，并将其保存的映射关系存入map里，我们便能轻易的启动其他module的Activity了。

- 第二节 路由框架的初始化

上节我们已经通过apt生成了映射文件，并且知道了如何通过映射文件去调用Activity，然而我们要实现一个路由框架，就要考虑在合适的时机拿到这些映射文件中的信息，以供上层业务做跳转使用。那么在什么时机去拿到这些映射文件中的信息呢？首先我们需要在上层业务做路由跳转之前

把这些路由映射关系拿到手，但我们不能事先预知上层业务会在什么时候做跳转，那么拿到这些路由关系最好的时机就是应用程序初始化的时候。

知道了在什么时机去拿到映射关系，接下来就要考虑如何拿了。我们在上面已经介绍过实现IRouteRoot接口的所有类文件里保存着各个module的分组文件(分组文件就是实现了IRouteGroup接口的类文件)，那么只要拿到所有实现IRouteGroup接口的类的集合，便能得到左右的路由信息了。下面看初始化的代码：

```
public class MyApplication extends Application {
 @Override public void onCreate() {
 super.onCreate(); EasyRouter.init(this); } // 我们手动实现的路由框架，我们就叫它EasyRouter
 public class EasyRouter {
 private static final String TAG = "EasyRouter";
 private static final String ROUTE_ROOT_PAKCAGE = "com.xsm.easyrouter.routes";
 private static final String SDK_NAME = "EaseRouter";
 private static final String SEPARATOR = "_";
 private static final String SUFFIX_ROOT = "Root";
 private static EasyRouter sInstance;
 private static Application mContext;
 private Handler mHandler;
 private EasyRouter() {
 mHandler = new Handler(Looper.getMainLooper()); }
 public static EasyRouter getInstance() {
 synchronized (EasyRouter.class) {
 if (sInstance == null) {
 sInstance = new EasyRouter(); }
 }
 return sInstance; }
 public static void init(Application application) {
 mContext = application;
 try {
 loadInfo();
 } catch (Exception e) {
 e.printStackTrace();
 Log.e(TAG, "初始化失败!", e); } } //...
 }
}
```

可以看到，进程启动的时候我们调用EasyRouter.init()方法，init()方法中调用了loadInfo()方法，而这个loadInfo()便是我们初始化的核心。我把loadInfo的代码贴出来：

```
private static void loadInfo() throws
PackageManager.NameNotFoundException,
InterruptedException, ClassNotFoundException,
NoSuchMethodException, IllegalAccessException,
InvocationTargetException, InstantiationException { //
获得所有 apt生成的路由类的全类名 (路由表) Set<String>
routerMap = classutils.getFileNameByPackageName(mContext,
ROUTE_ROOT_PAKCAGE); for (String className :
routerMap) { if
(className.startsWith(ROUTE_ROOT_PAKCAGE + "." + SDK_NAME
+ SEPARATOR + SUFFIX_ROOT)) { //root中注册的是分
组信息 将分组信息加入仓库中 ((IRouteRoot)
Class.forName(className).getConstructor().newInstance()).
loadInto(warehouse.groupsIndex); } } for
(Map.Entry<String, Class<? extends IRouteGroup>>
stringClassEntry : warehouse.groupsIndex.entrySet()) {
 Log.d(TAG, "Root映射表[" + stringClassEntry.getKey()
+ " : " + stringClassEntry.getValue() + "]"); }}}
```

我们首先通过ClassUtils.getFileNameByPackageName(mContext, ROUTE\_ROOT\_PAKCAGE)得到apt生成的所有实现IRouteRoot接口的类文件集合，通过上面的讲解我们知道，拿到这些类文件便可以得到所有的路由地址和Activity映射关系。

这个ClassUtils.getFileNameByPackageName()方法就是具体的实现了，下面我们看具体的代码：

想  
識  
思  
學

```
/** * 得到路由表的类名 * @param context *
@param packageName * @return * @throws
PackageManager.NameNotFoundException * @throws
InterruptedException */ public static Set<String>
getFileNameByPackageName(Application context, final
String packageName) throws
PackageManager.NameNotFoundException,
InterruptedException { final Set<String>
classNames = new HashSet<>(); List<String> paths =
getSourcePaths(context); //使用同步计数器判断均处理完成
 final CountDownLatch countDownLatch = new
CountDownLatch(paths.size()); ThreadPoolExecutor
threadPoolExecutor =
DefaultPoolExecutor.newDefaultPoolExecutor(paths.size());
 for (final String path : paths) {
threadPoolExecutor.execute(new Runnable() {
@Override public void run() {
DexFile dexFile = null; try {
//加载 apk中的dex 并遍历 获得所有包名为
{packageName} 的类 dexFile = new
DexFile(path); Enumeration<String>
dexEntries = dexFile.entries();
while (dexEntries.hasMoreElements()) {
String className = dexEntries.nextElement();
if (!TextUtils.isEmpty(className)
&& className.startsWith(packageName)) {
classNames.add(className);
}
}
}
catch (IOException e) {
e.printStackTrace(); } finally {
if (null != dexFile) {
try {
dexFile.close(); } catch
(IOException e) {
e.printStackTrace(); }
}
}
//释放一个
countDownLatch.countDown();
}
}); }
}
//
```

```
等待执行完成 countDownLatch.await(); return
classNames; }
```

这个方法会通过开启子线程，去扫描apk中所有的dex，遍历找到所有包名为packageName的类名，然后将类名再保存到classNames集合里。

List paths = getSourcePaths(context)这句代码会获得所有的apk文件(instant run会产生很多split apk),这个方法的具体实现大家看demo即可，不再阐述。这里用到了CountDownLatch类，会分path一个文件一个文件的检索，等到所有的类文件都找到后便会返回这个Set集合。所以我们可以知道，初始化时找到这些类文件会有一定的耗时，如果你已经看过ARouter的源码便会知道ARouter这里会有一些优化，只会遍历找一次类文件，找到之后就会保存起来，下次app进程启动会检索是否有保存这些文件，如果有就会直接调用保存后的数据去初始化。

- 第三节 路由跳转实现

经过上节的介绍，我们已经能够在进程初始化的时候拿到所有的路由信息，那么实现跳转便好做了。直接看代码：

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity { public void
startModule1MainActivity(View view) {
EasyRouter.getInstance().build("/module1/module1main").n
avigation(); }}
```

在build的时候，传入要跳转的路由地址，build()方法会返回一个Postcard对象，我们称之为跳卡。然后调用Postcard的navigation()方法完成跳转。用过ARouter的对这个跳卡都应该很熟悉吧！Postcard里面保存着跳转的信息。下面我把Postcard类的代码实现粘下来：

想  
識  
思  
學

```
public class Postcard extends RouteMeta { private
Bundle mBundle; private int flags = -1; //新版风格
private Bundle optionsCompat; //老版 private int
enterAnim; private int exitAnim; public
Postcard(String path, String group) { this(path,
group, null); } public Postcard(String path, String
group, Bundle bundle) { setPath(path);
setGroup(group); this.mBundle = (null == bundle ?
new Bundle() : bundle); } public Bundle getExtras()
{return mBundle;} public int getEnterAnim() {return
enterAnim;} public int getExitAnim() {return
exitAnim;} /** * 跳转动画 * @param enterAnim
* @param exitAnim * @return */ public Postcard
withTransition(int enterAnim, int exitAnim) {
this.enterAnim = enterAnim; this.exitAnim =
exitAnim; return this; } /** * 转场动画
* @param compat * @return */ public Postcard
withOptionsCompat(ActivityOptionsCompat compat) {
if (null != compat) { this.optionsCompat =
compat.toBundle(); } return this; }
public Postcard withString(@Nullable String key,
@Nullable String value) { mBundle.putString(key,
value); return this; } public Postcard
withBoolean(@Nullable String key, boolean value) {
mBundle.putBoolean(key, value); return this; }
 public Postcard withInt(@Nullable String key, int
value) { mBundle.putInt(key, value); return
this; } //还有许多给intent中bundle设置值得方法我就不一一
列出来了，可以看demo里所有的细节 public Bundle
getOptionsBundle() { return optionsCompat; }
public Object navigation() { return
EasyRouter.getInstance().navigation(null, this, -1,
null); } public Object navigation(Context context)
{ return
EasyRouter.getInstance().navigation(context, this, -1,
null); } public Object navigation(Context context,
NavigationCallback callback) { return
EasyRouter.getInstance().navigation(context, this, -1,
```

```
callback); } public Object navigation(Context context, int requestCode) { return EasyRouter.getInstance().navigation(context, this, requestCode, null); } public Object navigation(Context context, int requestCode, NavigationCallback callback) { return EasyRouter.getInstance().navigation(context, this, requestCode, callback); }}
```

如果你是一个Android开发，Postcard类里面的东西就不用我再给你介绍了吧！（哈哈）我相信你一看就明白了。我们只介绍一个方法navigation()，他有好几个重载方法，方法里面会调用EasyRouter类的navigation()方法。EaseRouter的navigation()方法，就是跳转的核心了。下面请看：



想  
識  
思  
學

```
protected Object navigation(Context context, final
Postcard postcard, final int requestCode, final
NavigationCallback callback) { try {
prepareCard(postcard); }catch (NoRouteNotFoundException
e) { e.printStackTrace(); //没找到 if
(null != callback) {
callback.onLost(postcard); } return null;
} if (null != callback) {
callback.onFound(postcard); } switch
(postcard.getType()) { case ACTIVITY:
final Context currentContext = null == context ? mContext
: context; final Intent intent = new
Intent(currentContext, postcard.getDestination());
 intent.putExtras(postcard.getExtras());
int flags = postcard.getFlags(); if (-1 !=
flags) { intent.setFlags(flags);
} else if (!(currentContext instanceof Activity)) {
 intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
} mHandler.post(new Runnable() {
@Override public void run() {
//可能需要返回码 if
(requestCode > 0) {
ActivityCompat.startActivityForResult((Activity)
currentContext, intent,
requestCode, postcard.getOptionsBundle());
} else {
ActivityCompat.startActivity(currentContext, intent,
postcard
.getOptionsBundle()); }
if ((0 != postcard.getEnterAnim() || 0 !=
postcard.getExitAnim()) &&
currentContext instanceof Activity) {
//老版本 ((Activity)
currentContext).overridePendingTransition(postcard
.getEnterAnim()
, postcard.getExitAnim());
}
//跳转完成
if (null != callback) {
```

```
callback.onArrival(postcard);
 }
 break;
default:
 break; } return null; }
```

这个方法里先去调用了prepareCard(postcard)方法,  
prepareCard(postcard)代码我贴出来,

链家网课

```

private void prepareCard(Postcard card) { RouteMeta
routeMeta = warehouse.routes.get(card.getPath()); if
(null == routeMeta) { Class<? extends IRouteGroup>
groupMeta = warehouse.groupsIndex.get(card.getGroup()); if (null == groupMeta) { throw new
NoRouteNotFoundException("没找到对应路由: 分组=" +
card.getGroup() + " 路径=" + card.getPath()); }
 IRouteGroup iGroupInstance; try {
iGroupInstance =
groupMeta.getConstructor().newInstance(); } catch
(Exception e) { throw new RuntimeException("路
由分组映射表记录失败.", e); }
 iGroupInstance.loadInto(warehouse.routes); //已经准
备过了就可以移除了 (不会一直存在内存中)
 warehouse.groupsIndex.remove(card.getGroup()); //再
次进入 else prepareCard(card); } else {
//类 要跳转的activity 或IService实现类
 card.setDestination(routeMeta.getDestination());
 card.setType(routeMeta.getType()); switch
(routeMeta.getType()) { case ISERVICE:
 Class<?> destination = routeMeta.getDestination();
 IService service =
warehouse.services.get(destination); if
(null == service) { try {
service = (IService)
destination.getConstructor().newInstance();
warehouse.services.put(destination, service);
 } catch (Exception e) {
e.printStackTrace();
 }
 card.setService(service);
 break;
 default:
 break;
 }
}
}

```

注意，Warehouse就是专门用来存放路由映射关系的类，里面保存着存路由信息的map，这在ARouter里面也是一样的。这段代码Warehouse.routes.get(card.getPath())通过path拿到对应的RouteMeta，这个RouteMeta里面保存了activityClass等信息。继续往下

看，如果判断拿到的RouteMeta是空，说明这个路由地址还没有加载到map里面(初始化时为了节省性能，只会加载所有的分组信息，而每个分组下的路由映射关系，会使用懒加载，在首次用到的时候去加载)，只有在第一次用到当前路由地址的时候，会去Warehouse.routes里面拿routeMeta，如果拿到的是空，会根据当前路由地址的group拿到对应的分组，通过反射创建实例，然后调用实例的loadInfo方法，把它里面保存的映射信息添加到Warehouse.routes里面，并且再次调用prepareCard(card)，这时再通过Warehouse.routes.get(card.getPath())就可以顺利拿到RouteMeta了。进入else{}里面，调用了card.setDestination(routeMeta.getDestination())，这个setDestination就是将RouteMeta里面保存的activityClass放入Postcard里面，下面switch代码块可以先不用看，这是实现ARouter中通过依赖注入实现Provider服务的逻辑，有心研究的同学可以去读一下demo。

好了，prepareCard()方法调用完成后，我们的postcard里面就保存了activityClass，然后switch (postcard.getType()){}会判断postcard的type为ACTIVITY，然后通过ActivityCompat.startActivity启动Activity。到这里，路由跳转的实现已经讲解完毕了。

## 小结

EaseRouter本身只是参照ARouter手动实现的路由框架，并且剔除掉了很多东西，如过滤器等，如果想要用在项目里，建议还是用ARouter更好，毕竟这只是个练手项目，功能也不够全面，当然有同学想对demo扩展后使用那当然更好，遇到什么问题可以及时联系我。我的目的是通过自己手动实现路由框架来加深对知识的理解，如这里面涉及到的知识点apt、javapoet和组件化思路、编写框架的思路等。看到这里，如果感觉干货很多，欢迎关注我的github，里面会有更多干货！

## demo地址

[仿ARouter一步步实现一个路由框架，点我访问源码，欢迎star](#)

### 20.3.2 ARouter组件化框架原理分析

ARouter是阿里巴巴开源的组件化架构框架，能帮助组件化项目中实现不同模块间的跳转，以及AOP面向切面的编程，能对页面跳转的过程进行很好的干预。本文将从源码角度入手，对该框架的原理进行分析。

项目集成时会集成两个Library，也对应了ARouter的两个阶段。arouter-compiler是用于编译期的，而arouter-api是面向运行期的。下面就从这两个阶段开始讲起。

```
dependencies { // Replace with the latest version
compile 'com.alibaba:arouter-api:?’
annotationProcessor 'com.alibaba:arouter-compiler:?’
...}
```

#### 编译阶段

ARouter是可以自动注册页面映射关系的，在每个目标页面上使用注解来标注一些参数，比方Path标注其路径。使用注解时会遇到的第一个问题就是需要找到处理注解的时机，如果在运行期处理注解则会大量地运用反射，而这在软件开发中是非常不合适的，因为反射本身就存在性能问题，如果大量地使用反射会严重影响APP的用户体验，而又因为路由框架是非常基础的框架，所以大量使用反射也会使得跳转流程的用户体验非常差。所以ARouter最终使用的方式是在编译期处理被注解的类，这样就可以做到在运行中尽可能不使用反射，这就是注解处理器的作用。

页面注册的整个流程首先通过注解处理器扫出被标注的类文件，然后按照不同种类的源文件进行分类，分别生成固定格式的类文件，命名规则是工程名ARouter+\$\$ +xxx+\$ \$ +模块名。可以看出这里面包含了Group、Interceptor、Providers以及Root。这部分完成之后就意味着编译期的工作已经结束了，之后的初始化其实是发生在运行期的，在运行期只需要通过固定的包名来加载映射文件就可以了，因为注解是由开发者自己完成的，所以了解其中的规则，就可以在使用的时候利用相应的规则反向地提取出来。这就是页面自动注册的整个流程。

自动生成的类：

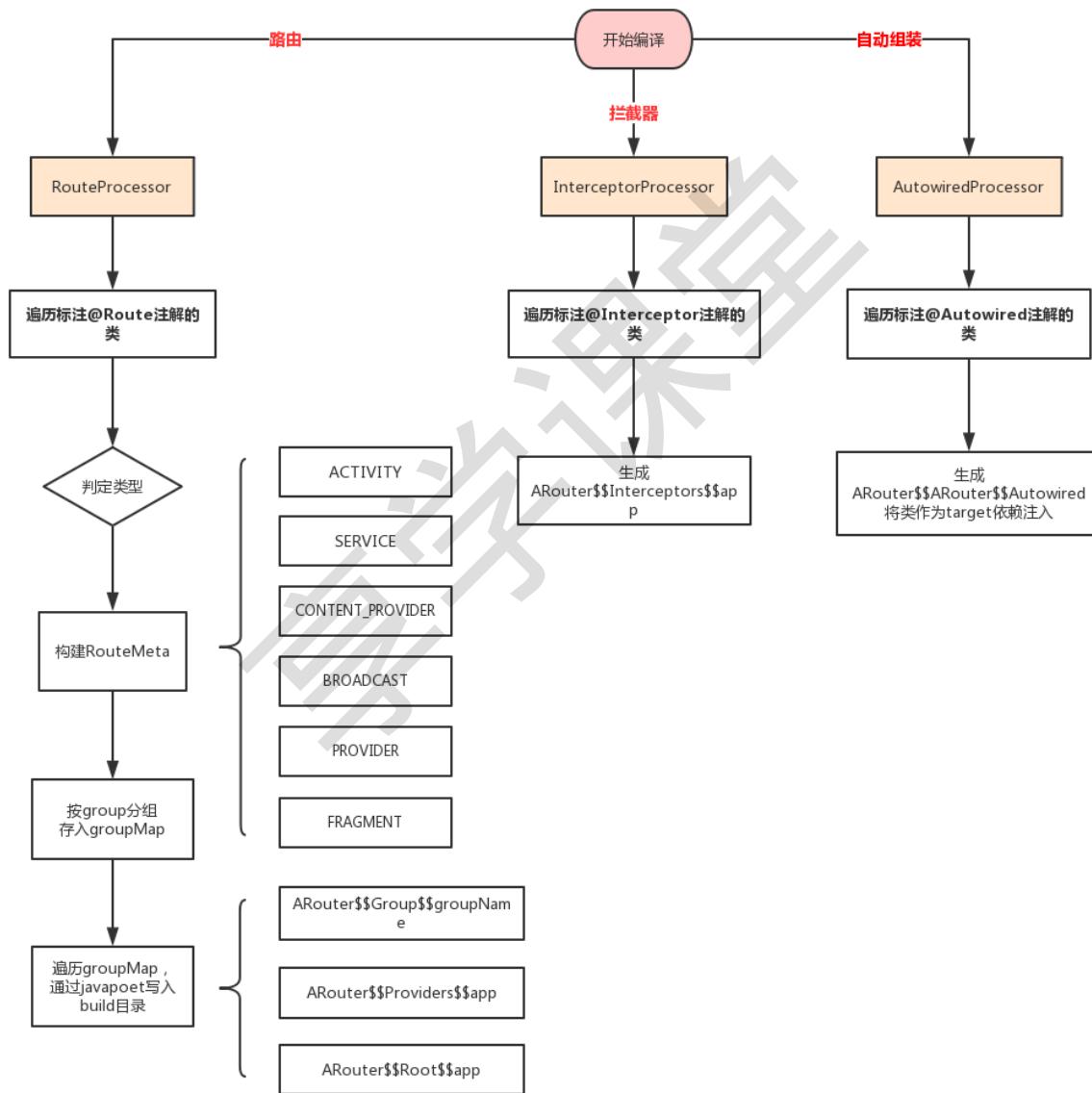
```

 < generated
 > not_namespaced_r_class_sources
 > res
 < source
 < apt
 < debug
 < com.alibaba.android.arouter.routes
 < ARouter$$Group$$moudle
 < ARouter$$Interceptors$$modulea
 < ARouter$$Providers$$modulea
 < ARouter$$Root$$modulea
 > buildConfig
 > buildConfig
 > buildConfig

```

<https://blog.csdn.net/lbcwnu>

## 编译期流程图：



## 运行阶段

运行阶段也分为两部分来分析，初始化和页面跳转。

### 初始化

```
// Initialize the SDK ARouter.init(mApplication);
// As early as possible, it is recommended to initialize
in the Application //代码流程 ...
ARouter.init(mApplication) -> _ARouter.init(application)
-> LogisticsCenter.init(mContext, executor) ...
```

沿着代码流程init会走到LogisticsCenter.init(mContext, executor)，该函数通过ClassUtils.getFileNameByPackageName遍历包名"com.alibaba.android.arouter.routes"下的所有class存入routerMap中（即把所有编译期自动生成的class读取出来），然后通过for循环把这些class按照不同类型（Root, Interceptors, Providers）进行处理。通过class的路径，利用Class.forName的方式得到class实例，然后调用该class的loadInto方法把该class关联到Warehouse(Warehouse是整个项目的仓库，里面存有所有class的映射关系)的相应的结构体中,这样所有自动生成的映射关系就存储在内存中了，以便于后续查找时使用。

想  
識  
思  
學

```
/** * LogisticsCenter init, load all metas in
memory. Demand initialization */ public
synchronized static void init(Context context,
ThreadPoolExecutor tpe) throws HandlerException {
mContext = context; executor = tpe; try {
 if (registerByPlugin) {
logger.info(TAG, "Load router map by arouter-auto-
register plugin."); } else {
Set<String> routerMap; // It will rebuild
router map every times when debuggable. if
(ARouter.debuggable() ||
PackageUtils.isNewVersion(context)) {
logger.info(TAG, "Run with debug mode or new install,
rebuild router map."); // These class
was generated by arouter-compiler.
routerMap = Classutils.getFileNameByPackageName(mContext,
ROUTE_ROOT_PAKCAGE); if
(!routerMap.isEmpty()) {
context.getSharedPreferences(AROUTER_SP_CACHE_KEY,
Context.MODE_PRIVATE).edit().putStringSet(AROUTER_SP_KEY_
MAP, routerMap).apply(); }
 PackageUtils.updateVersion(context); // Save
new version name when router map update finishes.
 } else { logger.info(TAG, "Load
router map from cache."); routerMap =
new HashSet<>
(context.getSharedPreferences(AROUTER_SP_CACHE_KEY,
Context.MODE_PRIVATE).getStringSet(AROUTER_SP_KEY_MAP,
new HashSet<String>())); }
logger.info(TAG, "Find router map finished, map size = "
+ routerMap.size() + ", cost " +
(System.currentTimeMillis() - startInit) + " ms.");
 startInit = System.currentTimeMillis();
 for (String className : routerMap) {
 if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT +
SDK_NAME + SEPARATOR + SUFFIX_ROOT)) {
 // This one of root elements, load root.
 ((IRouteRoot)
```

```
(Class.forName(className).getConstructor().newInstance())
).loadInto(warehouse.groupsIndex); }
else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT +
SDK_NAME + SEPARATOR + SUFFIX_INTERCEPTORS)) {
 // Load interceptorMeta
 ((IInterceptorGroup)
(Class.forName(className).getConstructor().newInstance())
).loadInto(warehouse.interceptorsIndex);
} else if (className.startsWith(ROUTE_ROOT_PAKCAGE +
DOT + SDK_NAME + SEPARATOR + SUFFIX_PROVIDERS)) {
 // Load providerIndex
 ((IProviderGroup)
(Class.forName(className).getConstructor().newInstance())
).loadInto(warehouse.providersIndex);
}
}
logger.info(TAG, "Load root element finished, cost " +
(System.currentTimeMillis() - startInit) + " ms.");
if (warehouse.groupsIndex.size() == 0) {
 logger.error(TAG, "No mapping files were found, check
your configuration please!"); }
if (ARouter.debuggable()) {
 logger.debug(TAG,
String.format(Locale.getDefault(), "LogisticsCenter has
already been loaded, GroupIndex[%d],
InterceptorIndex[%d], ProviderIndex[%d]",
Warehouse.groupsIndex.size(),
Warehouse.interceptorsIndex.size(),
Warehouse.providersIndex.size())); }
catch (Exception e) {
 throw new
HandlerException(TAG + "ARouter init logistics center
exception! [" + e.getMessage() + "]"); }
}
```

细心的读者可能发现了，这里加载了Root分组，却没有加载Group分组，那么为什么要这样设计呢，这两者又有什么关系呢？这里就涉及ARouter的一个加载理念：分组管理，按需加载。

## 加载理念

如果一个App有上百个页面的时候，一次性将所有页面都加载到内存中本身对于内存的损耗是非常可怕的，同时对于性能的损耗也是不可忽视的。所以ARouter中提出了分组的概念，ARouter允许某一个模块下有多个分组，所有的分组最终会被一个root节点管理。如下图中所示，假设有4个模块，每个模块下面都有一个root结点，每个root结点都会管理整个模块中的group节点，每个group结点则包含了该分组下的所有页面，也就是说可以按照一定的业务规则或者命名规范把一部分页面聚合成一个分组，每个分组其实就相当于路径中的第一段，而每个模块中都会有一个拦截器节点就是Interceptor结点，除此之外每个模块还会有控制拦截反转的provider结点。



ARouter在初始化的时候只会一次性地加载所有的Root结点，而不会加载任何一个Group结点，这样就会极大地降低初始化时加载结点的数量。那么什么时候加载分组结点呢？其实就是当某一个分组下的某一个页面第一次被访问的时候，整个分组的全部页面都会被加载进去，这就是ARouter的按需加载。其实在整个APP运行的周期中，并不是所有的页面都需要被访问到，可能只有20%的页面能够被访问到，所以这时候使用按需加载的策略就显得非常重要的，这样就会减轻很大的内存压力。

至此初始化阶段就分析结束了，下面看跳转阶段。

## 跳转

我们也从跳转代码说起，通常的跳转代码如下。

```
ARouter.getInstance().build("/moudle/activitya").navigation();...//代码流程build("path") ->
_ARouter.getInstance().build(path) -> build(String path,
String group) -> new Postcard(path, group) ->
Postcard.navigation() ->
LogisticsCenter.completion(postcard) ->
_navigation(context, postcard, requestCode, callback);...
```

\_ARouter.getInstance().build(path)源码，构建并返回了一个Postcard结构，该结构的定义是A container that contains the roadmap (包含 roadmap的一个容器)。

函数首先去查找PathReplaceService.class接口的实现类，该实现类的作用就是实现“运行期动态修改路由”，如果找到则利用forString方法修改path。如没有此类则返回build(path, extractGroup(path))，其中extractGroup()是从路径中获取默认的分组信息。最后根据提供的path和group创建一个Postcard对象返回。

```
/** * Build postcard by path and default group
 */ protected Postcard build(String path) { if (TextUtils.isEmpty(path)) { throw new HandlerException(Consts.TAG + "Parameter is invalid!"); } else { PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class); if (null != pService) { path = pService.forString(path); } } return build(path, extractGroup(path)); }
```

接下来会调用Postcard.navigation()方法。该函数首先拼装了postcard结构体，并判断是否需要拦截器功能，如果不需要则直接调用\_navigation()进行跳转了。

想  
識  
思  
學

```
/** * Use router navigation. * * @param
context Activity or null. * @param postcard
Route metas * @param requestCode RequestCode *
@param callback cb */ protected Object
navigation(final Context context, final Postcard
postcard, final int requestCode, final NavigationCallback
callback) { try {
LogisticsCenter.completion(postcard); } catch
(NoRouteFoundException ex) {
logger.warning(Consts.TAG, ex.getMessage());
if (debuggable()) { // Show friendly tips
for user. runInMainThread(new Runnable() {
@Override
public
void run() {
Toast.makeText(mContext, "There's no route matched!\n" +
 " Path = [" +
postcard.getPath() + "]\n" +
 " Group = [" + postcard.getGroup() + "]", Toast.LENGTH_LONG).show();
}
} if (null != callback) {
callback.onLost(postcard);
} else { // No callback for this invoke, then we use the
global degrade service. DegradeService
degradeService =
ARouter.getInstance().navigation(DegradeService.class);
if (null != degradeService) {
degradeService.onLost(context, postcard);
}
} return null;
}
if (null != callback) {
callback.onFound(postcard); } //绿色通道校验 需要
拦截处理 if (!postcard.isGreenChannel()) { // It
must be run in async thread, maybe interceptor cost too
much time made ANR. //调用拦截器截面控制器，遍历内存
仓库的自定义拦截器，并在异步线程中执行拦截函数
interceptorService.doInterceptions(postcard, new
InterceptorCallback() { /**
* Continue process
*/
@param postcard route meta
*/
```

```
 @Override public void
onContinue(Postcard postcard) {
 _navigation(context, postcard, requestCode, callback);
 } /** *
Interrupt process, pipeline will be destory when this
method called. * * @param
exception Reson of interrupt. */
 @Override public void
onInterrupt(Throwable exception) { if
(null != callback) {
 callback.onInterrupt(postcard); }
 logger.info(Consts.TAG, "Navigation
failed, termination by interceptor : " +
exception.getMessage()); } });
 } else { return _navigation(context,
postcard, requestCode, callback); } return
null; }
```

其中两个主要的方法是LogisticsCenter.completion()和\_navigation()。completion()里面填充postcard里面剩下的信息，在这里可以看到，如果该class没有被加载（即上面说的延时加载），在这里会把该模块下的所有类加载起来。最终得到完整的postcard结构体。

想  
識  
思  
學

```
/** * Completion the postcard by route metas
* * @param postcard Incomplete postcard, should
complete by this method. */
public synchronized
static void completion(Postcard postcard) {
 RouteMeta routeMeta =
Warehouse.routes.get(postcard.getPath()); //第一次跳
转时没有被加载，延时加载的地方 if (null == routeMeta) {
 // Maybe its does't exist, or didn't load.
 Class<? extends IRouteGroup> groupMeta =
Warehouse.groupsIndex.get(postcard.getGroup()); // Load
route meta. if (null == groupMeta) {
 throw new NoRouteNotFoundException(TAG + "There is no
route match the path [" + postcard.getPath() + "], in
group [" + postcard.getGroup() + "]"); } else
{
 // Load route and cache it into memory,
then delete from metas. try {
 IRouteGroup iGroupInstance =
groupMeta.getConstructor().newInstance();
 iGroupInstance.loadInto(Warehouse.routes);
 warehouse.groupsIndex.remove(postcard.getGroup());
 if (ARouter.debuggable()) {
 logger.debug(TAG,
String.format(Locale.getDefault(), "The group [%s] has
already been loaded, trigger by [%s]",
postcard.getGroup(), postcard.getPath()));
 } } catch (Exception e) {
 throw new HandlerException(TAG + "Fatal exception
when loading group meta. [" + e.getMessage() + "]");
 } completion(postcard); // Reload
} } else {
postcard.setDestination(routeMeta.getDestination());
 postcard.setType(routeMeta.getType());
postcard.setPriority(routeMeta.getPriority());
postcard.setExtra(routeMeta.getExtra()); Uri
rawUri = postcard.getUri(); if (null !=
rawUri) { // Try to set params into bundle.
 Map<String, String> resultMap =
TextUtils.splitQueryParameters(rawUri);
```

```
Map<String, Integer> paramsType =
 routeMeta.getParamsType(); //
 if
 (MapUtils.isNotEmpty(paramsType)) {
 Set value by its type, just for params which annotation
 by @Param
 for (Map.Entry<String,
Integer> params : paramsType.entrySet()) {
 setvalue(postcard,
 params.getValue(),
 params.getKey(),
 resultMap.get(params.getKey()));
 }
 // Save params name which need auto
 inject.
 postcard.getExtras().putStringArray(ARouter.AUTO_INJECT,
 paramsType.keySet().toArray(new String[]{}));
 }
 // Save raw uri
 postcard.withString(ARouter.RAW_URI, rawUri.toString());
}
```

\_navigation是真正实现跳转的地方，通过postcard.getDestination()获取目标class的实例（在初始化时已经加载到内存），用大家熟悉的startactivity方法进行了跳转。

想  
識  
思  
學

```
 private Object _navigation(final Context context,
final Postcard postcard, final int requestCode, final
NavigationCallback callback) { final Context
currentContext = null == context ? mContext : context;
 switch (postcard.getType()) { case
ACTIVITY: // Build intent
final Intent intent = new Intent(currentContext,
postcard.getDestination());
intent.putExtras(postcard.getExtras()); //
Set flags. int flags =
postcard.getFlags(); if (-1 != flags) {
 intent.setFlags(flags); }
else if (!(currentContext instanceof Activity)) { // Non activity, need less one flag.
 intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
 } // Set Actions
String action = postcard.getAction(); if
(!TextUtils.isEmpty(action)) {
 intent.setAction(action); }
// Navigation in main looper.
runInMainThread(new Runnable() {
@Override public void run() {
 startActivity(requestCode, currentContext,
intent, postcard, callback); }
); break; case
PROVIDER: return postcard.getProvider();
 case BROADCAST: case
CONTENT_PROVIDER: case FRAGMENT:
Class fragmentMeta = postcard.getDestination();
try { Object instance =
fragmentMeta.getConstructor().newInstance();
if (instance instanceof Fragment) {
 ((Fragment)
instance).setArguments(postcard.getExtras());
 } else if (instance instanceof
android.support.v4.app.Fragment) {
 ((android.support.v4.app.Fragment)
instance).setArguments(postcard.getExtras());
 }
 }
 }
}
```

```
 }
 return instance;
 } catch (Exception ex) {
 logger.error(Consts.TAG, "Fetch fragment instance error,
" + TextUtils.formatStackTrace(ex.getStackTrace()));
 }
 case METHOD:
 case SERVICE:
 default:
 return null;
 }
 return null;
}
```

至此整个源码分析结束。

### 20.3.3 Android之ARouter使用和原理解析

#### ARouter使用和原理解析

github: <https://github.com/alibaba/ARouter>

##### 一：ARouter介绍

组件化逐渐成为热潮，组件化可以使业务逻辑高度解耦、模块分离，提高开发效率，更有利于多人协作并行开发。组件化中两个单向依赖的module之间互相启动对方的Activity，因为没有相互引用startActivity()是实现不了的，必须需要一个协定的通信方式，此时ARouter路由框架就派上用场了。ARouter是阿里巴巴开源路由框架，主要解决组件间、模块间的界面跳转问题。

##### 二：ARouter基础功能使用

###### 1):添加依赖和配置

java和kotlin配置不一样，本文以kotlin和gradle3.4+进行配置的。

在module中build.gradle配置

```
apply plugin: 'kotlin-kapt' kapt {
 arguments {
 arg("AROUTER_MODULE_NAME", project.getName())
 }
 dependencies {
 implementation 'com.alibaba:arouter-api:1.5.0'
 kapt 'com.alibaba:arouter-compiler:1.2.2'
 }
}
```

###### 2)::在目标页面Activity添加注解

```
// 在支持路由的页面上添加注解(必选) // 这里的路径需要注意的是至少需要有两级, /xx/xx
```

```
@Route(path = "/test/activity")public class YourActivity
extend Activity { ... }
```

### 3): 初始化SDK

```
/** * 初始化ARouter */ private fun
initARouter() { if (BuildConfig.DEBUG) { //如果在
debug模式下 // 打印日志,默认关闭
ARouter.openLog() // 开启调试模式, 默认关闭(如果在
InstantRun模式下运行, 必须开启调试模式! 线上版本需要关闭,否则有安全
风险) ARouter.openDebug() // 打印日志
的时候打印线程堆栈 ARouter.printStackTrace()
} // 尽可能早, 推荐在Application中初始化
ARouter.init(this) }
```

### 4.发起路由操作

```
// 1. 应用内简单的跳转(通过URL跳转在'进阶用法'中)
ARouter.getInstance().build("/test/activity").navigation()
// 2. 跳转并携带参数
ARouter.getInstance().build("/test/1") .withLong("key1",
666L) .withString("key3", "888").withObject("key4", new
Test("Jack", "Rose")) .navigation();
```

## 三: ARouter原理

### 1):@Route注解的作用

在需要对其他module提供调用的Activity中, 都需要声明@Route注解, 我们称之为路由地址。

```
@Route(path = "/userCenter/login")
```

```
class LoginActivity : AppCompatActivity(){...}
```

那么这个注解有什么用呢?

路由框架会在项目的编译期通过注解处理器扫描所有@Route注解的Activity类，然后将Route注解中的path地址和Activity.class文件的映射关系保存到它自己生成的java文件的map中。

路径：

build/generated/source/kapt/debug/com/alibaba/android/arouter

伪代码：

```
public class ARouter$$Root$$app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("account",
 ARouter$$Group$$account.class); routes.put("shop",
 ARouter$$Group$$shop.class); }
 public class ARouter$$Group$$account implements IRouteGroup {
 @Override public void loadInto(Map<String, RouteMeta>
 atlas) { atlas.put("/account/order",
 RouteMeta.build(RouteType.ACTIVITY,
 RegionSelectActivity.class, "/account/order", "account",
 null, -1, -2147483648)); }
 }
}
```

2):ARouter如何实现跳转

```
//用ARouter启动Activity，代码如下：
ARouter.getInstance().build("/userCenter/login").navigation(); //实现伪代码：
public void login(String name, String password) {
 HashMap<String, ClassBean> route = MyRouters.getRouteInfo(new HashMap<String, ClassBean>());
 LoginActivity.class classBean = route.get("/userCenter/login"); Intent intent =
 new Intent(this, classBean); intent.putExtra("name", name); intent.putExtra("password", password);
 startActivity(intent); }
```

总结：

1、我们通过@Route，将path和目标类进行关联。

2、在编译阶段，注解处理器RouteProcessor会在com.alibaba.android.arouter.routes包下以固定的拼接规则生成ARouter\$\$Root\$\$app和多个ARouter\$\$Group\$\$xxx文件。以map的形式，在group文件中存储对应的path和目标类信息，在root文件中存储groupName和对应的group类的class对象。

3、在application中调用init进行初始化。

4、在init方法中，先调用loadRouterMap方法。这是一个空实现，是给插件插桩用的。我们可以配置Gradle插件实现路由表的自动加载，代替扫描dex文件的方式加载路由表。缩短初始化时间和解决应用加固无法直接访问dex文件的问题。

arouter-auto-register插件将在此方法内部生成代码，调用此方法以注册所有路由器，拦截器和提供程序。在这里会对registerByPlugin置为true。这样就避免扫描dex文件。

5、如果没有配置gradle插件实现路由表的自动加载，我们就需要自己扫描dex文件。

6、获取全局路由信息是一个比较耗时的操作，所以ARouter就通过将全局路由信息缓存到SP中来实现复用。

7、如果当前开启了debug模式或者通过本地SP缓存判断出app的版本前后发生了变化，那么就重新获取全局路由信息，否则就还是使用之前缓存到SP中的数据。

8、通过dex扫描com.alibaba.android.arouter.routes包下的文件，然后将文件的全路径存储到HashSet中，然后存到sp中。也就是说sp存储了在com.alibaba.android.arouter.routes这个包下自动生成的辅助文件的全路径。

9、通过判断路径名的前缀字符串，就可以知道该类文件对应什么类型，然后通过反射构建不同类型的对象，通过调用对象的loadInto方法将路由信息存到Warehouse的Map中。

10、对于大型App来说，一次性加载路由信息到内存，会占据大量内存。所以是按需加载。

11、如果说有/account/userHome，首先通过反射创建root类，调用loadInto方法将键值对保存到groupIndex的HashMap中。

12、当后续跳转到group为account的页面时，再通过反射调用account类的loadInto方法，将键值对存到routes的HashMap中。实现了按需加载

#### 四:ARouter进阶用法

1):跳转界面不带参

2):跳转界面带参

3):跳转界面传递对象

4):Uri跳转

5):跳转结果监听

6):声明拦截器（拦截跳转过程，面向切面编程）

7):为目标页面声明更多信息

8):依赖注入解耦

#### 五、ARouter实现参数注入

- 控制反转：通过接口获得实现类。以一定的拼接规则，通过反射的反射获得实现类。
- 通过setIntent进行赋值
- 注解处理器会在编译阶段扫描有被Autowired注解标注的变量，根据这个类和变量的情况生成一份Java类Autowired。这个类实现了IProvider接口，实现inject方法，在inject方法中通过getIntent获取值，然后对注解变量进行赋值。这里用到了控制反转，通过接口IProvider获得实现类。
- 我们在onCreate调用ARouter.getInstance().inject(this)方法，在inject中通过一定的拼接规则获取到上文所述的Java类ARouter\$\$Autowired的className，通过反射创建类对象。后调用对象的inject方法，完成对数据的赋值。

```
ISyringe autowiredHelper = (ISyringe)
Class.forName(instance.getClass().getName() +
SUFFIX_AUTOWIRED).getConstructor().newInstance(); autowire
dHelper.inject(instance);
```

补充：

Q1：组件化中模块之间的Activity跳转，为什么选择ARouter路由跳转

1):隐式跳转,但是一个项目中不可能所有的跳转都是隐式的，这样Manifest文件会有很多过滤配置，而且非常不利于后期维护。

2):反射拿到Activity的class文件也可以实现跳转，

缺点:1大量的使用反射跳转对性能会有影响，

2.需要拿到Activity的类文件，在组件开发的时候，想拿到其他module的类文件是很麻烦的，因为组件开发的时候组件module之间是没有相互引用的，你只能通过找到类的路径去反射拿到这个class。

Q2：什么是APT

APT是Annotation Processing Tool的简称,即注解处理工具。apt是在编译期对代码中指定的注解进行解析，然后做一些其他处理（如通过javapoet生成新的Java文件）。我们常用的ButterKnife，其原理就是通过注解处理器在编译期扫描代码中加入的@BindView、@OnClick等注解进行扫描处理，然后生成XXX\_ViewBinding类，实现了view的绑定。

#### 20.3.4 手动实现一个路由框架EasyRouter

一、从组件化引入路由设计需要满足的条件

为什么在这里要介绍组件化呢？因为路由天生契合组件化。并不是说路由是为组件化而设计的，实际上两者没有任何关系。

**组件化需要路由设计满足的条件**

标题很绕口，组件化是最近比较流行的架构思想，组件化使业务逻辑高度的解耦、模块分离等，同时也使多人协作并行开发变得更加简单，每个人只需要负责自己的模块就好。下面我放一张图来使大家更好地理解：

如上图，在组件化中，为了业务逻辑的彻底解耦，同时也为了每个 module都可以方便的单独运行和调试，上层的各个module不会进行相互依赖(只有在正式联调的时候才会让app壳module去依赖上层的其他组件 module)，而是共同依赖于base module，base module中会依赖一些公共的第三方库和其他配置。

在开发阶段，我们的组件模块都是application的形式存在的，各个模块单独编译和运行，互不依赖，而在项目正式合并打线上包的时候，又要求组件模块以library的形式被主模块依赖。那么在开发阶段上层的各个 module中，如何进行通信呢？

我们知道，传统的Activity之间通信，通过startActivity(intent)，而在组件化的项目中，上层的module没有依赖关系(即便两个module有依赖关系，也只能是单向的依赖)，那么假如login module中的一个Activity需要启动pay\_module中的一个Activity便不能通过startActivity来进行跳转。那么大家想一下还有什么其他办法呢？可能有同学会想到隐式跳转，这当然也是一种解决方法，但是一个项目中不可能所有的跳转都是隐式的，这样Manifest文件会有很多过滤配置，而且非常不利于后期维护。当然你用反射拿到Activity的class文件也可以实现跳转，但是第一：大量的使用反射跳转对性能会有影响，第二：你需要拿到Activity的类文件，在组件开发的时候，想拿到其他module的类文件是很麻烦的，因为组件开发的时候组件module之间是没有相互引用的，你只能通过找到类的路径去反射拿到这个class，那么有没有一种更好的解决办法呢？办法当然是有的。下面看图：

在组件化中，我们通常都会在base\_module上层再依赖一个 router\_module,而这个router\_module就是负责各个模块之间页面跳转的。

用过ARouter路由框架的同学应该都知道，在每个需要对其他module提供调用的Activity中，都会声明类似下面@Route注解，我们称之为路由地址：

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity { @Override protected
void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main); } }
@Route(path = "/module1/module1main")public class Module1MainActivity
extends AppCompatActivity { @Override protected
void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_module1_main); } }
```

那么这个注解有什么用呢，路由框架会在项目的编译器扫描所有添加 @Route注解的Activity类，然后将route注解中的path地址和 Activity.class文件一一对应保存，如直接保存在map中。为了让大家理解，我这里来使用近乎伪代码给大家简单演示一下。

```
//项目编译后通过apt生成如下方法public HashMap<String,
ClassBean> routeInfo() { HashMap<String, ClassBean>
route = new HashMap<String, ClassBean>();
route.put("/main/main", MainActivity.class);
route.put("/module1/module1main",
Module1MainActivity.class); route.put("/login/login",
LoginActivity.class);}
```

这样我们想在app模块的MainActivity跳转到login模块的LoginActivity，那么便只需调用如下：

```
//不同模块之间启动Activitypublic void login(String name,
String password) { HashMap<String, ClassBean> route =
routeInfo(); LoginActivity.class classBean =
route.get("/login/login"); Intent intent = new
Intent(this, classBean); intent.putExtra("name",
name); intent.putExtra("password", password);
startActivity(intent);}
```

用过ARouter的同学应该知道，用ARouter启动Activity应该是下面这个写法

```
// 2. Jump with
parametersARouter.getInstance().build("/test/login")
.withString("password", 66666)
.withString("name", "小三") .navigation();
```

那么ARouter背后的原理是怎么样的呢？实际上它的核心思想跟上面讲解的是一样的，我们在代码里加入的@Route注解，会在编译时期通过apt生成一些存储path和activityClass映射关系的类文件，然后app进程启动的时候会拿到这些类文件，把保存这些映射关系的数据读到内存里(保存在map里)，然后在进行路由跳转的时候，通过build()方法传入要到达页面的路由地址，ARouter会通过它自己存储的路由表找到路由地址对应的Activity.class(activity.class = map.get(path))，然后new Intent()，当调用ARouter的withString()方法它的内部会调用intent.putExtra(String name, String value)，调用navigation()方法，它的内部会调用startActivity(intent)进行跳转，这样便可以实现两个相互没有依赖的module顺利的启动对方的Activity了。

## 路由的多种用途

实际上，路由不是为组件化而生，因为它的用处很多，比如配合webview中打开客户端原生页面，配合后台打开不同的页面等。而类似ARouter这种路由呢，还有另一个功能，就是业务解耦后各个业务线的相互调用，我将它理解为业务路由，比如支付模块需要用到用户模块的用户信息，那么为了解耦，用户模块只会给外部声明调用用户信息的方法，并不关心谁要用到用户信息。而支付模块看到你的声明，就知道你有获取用户信息这个功能，调用即可。而这个功能的实现就需要依赖ARouter中的依赖注入了，我们最后几节会说到。

## 二、通过Route注解去探究如何实现路由跳转

上节大概了解了根据路由地址去实现跳转的方式，只是做个启发，这节我们就详细介绍一下实现的细节。

### 通过Route注解去探究如何实现路由跳转

简单讲，要通过Route注解生成我们的路由表，首先第一步需要定义注解

```
@Target(ElementType.TYPE)@Retention(RetentionPolicy.CLASS)
public @interface Route { /** * 路由的路径 *
@return */ String path(); /** * 将路由节点进行
分组，可以实现动态加载 * @return */ String group()
default "";}
```

这里看到Route注解里有path和group，这便是仿照ARouter对路由进行分组。因为当项目变得越来越大庞大的时候，为了便于管理和减小首次加载路由表过于耗时的问题，我们对所有的路由进行分组。在ARouter中会要求路由地址至少需要两级，如"/xx/xx",一个模块下可以有多个分组。这里我们就将路由地址定为必须大于等于两级，其中第一级是group。如app module下的路由注解：

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity {}@Route(path =
"/main/main2")public class Main2Activity extends
AppCompatActivity {}@Route(path = "/show/info")public
class ShowActivity extends AppCompatActivity {}
```

在项目编译的时候，我们将会通过apt生成EaseRouter\_Root\_app文件和EaseRouter\_Group\_main、EEaseRouter\_Group\_show等文件，EaseRouter\_Root\_app文件对应于app module，里面记录着本module下所有的分组信息，EaseRouter\_Group\_main、EaseRouter\_Group\_show文件分别记载着app module下每个分组的所有路由地址和ActivityClass映射信息。

本demo在编译的时候会生成类如下所示，先不要管这些类是怎么生成的，仔细看类的内容

```
public class EaseRouter_Root_app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("main",
 EaseRouter_Group_main.class); routes.put("show",
 EaseRouter_Group_show.class); } } public class
 EaseRouter_Group_main implements IRouteGroup { @Override
 public void loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/main/main", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, Main2\Activity.class, "/main/main", "main"));
 atlas.put("/main/main2", RouteMeta.build(RouteMeta.Type.AC
 TIVITY, Main2\Activity.class, "/main/main2", "main"));
 } } public class EaseRouter_Group_show implements
 IRouteGroup { @Override public void
 loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/show/info", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, ShowActivity.class, "/show/info", "show")); } }
```

大家会看到生成的类分别实现了IRouteRoot和IRouteGroup接口，并且实现了loadInto()方法，而loadInto方法通过传入一个特定类型的map就能把分组信息放入map里。这两个接口是干嘛的我们先搁置，继续往下看如果我们在login\_module中想启动app\_module中的MainActivity类，首先，我们已知MainActivity类的路由地址是"/main/main"，第一个"/main"代表分组名，那么我们岂不是可以像下面这样调用去得到MainActivity类文件，然后startActivity。这里的RouteMeta只是存有Activity class文件的封装类，先不用理会。

```
public void test() { EaseRouter_Root_app rootApp = new
EaseRouter_Root_app(); HashMap<String, Class<? extends
IRouteGroup>> rootMap = new HashMap<>();
rootApp.loadInto(rootMap); //得到/main分组 Class<?
extends IRouteGroup> aClass = rootMap.get("main"); try
{ HashMap<String, RouteMeta> groupMap = new
HashMap<>();
aClass.newInstance().loadInto(groupMap); //得到
MainActivity RouteMeta main =
groupMap.get("/main/main"); Class<?>
mainActivityClass = main.getDestination(); Intent
intent = new Intent(this, mainActivityClass);
startActivity(intent); } catch (InstantiationException
e) { e.printStackTrace(); } catch
(IllegalAccessException e) { e.printStackTrace();
}}}
```

可以看到，只要有了这些实现了IRouteRoot和IRouteGroup的类文件，我们便能轻易的启动其他module的Activity了。其实这就是路由跳转的原理，这些类文件，我们可以约定好之后，在代码的编写过程中自己手动实现，也可以通过apt生成。作为一个框架，当然是自动解析Route注解然后生成这些类文件更好了。要想自动生成这些路由地址与Activity的映射关系，那么便要了解apt和javapoet了。

### 三、利用apt和javapoet生成路由映射文件

通过上节我们知道在Activity类上加上@Route注解之后，便可通过apt来生成对应的路由表，那么这节我们就来讲述一下如何通过apt来生成路由表。这节我会拿着demo里面的代码来跟大家详细介绍，我们先来了解一下apt吧！

### 利用apt和javapoet生成映射文件

APT是Annotation Processing Tool的简称,即注解处理工具。它是在编译期对代码中指定的注解进行解析,然后做一些其他处理(如通过javapoet生成新的Java文件)。我们常用的ButterKnife,其原理就是通过注解处理器在编译期扫描代码中加入的@BindView、@OnClick等注解进行扫描处理,然后生成XXX\_ViewBinding类,实现了view的绑定。

关于apt和javapoet的文章,网上一大堆,下面我通过apt在路由框架中的应用来一步步解释它们的用法。

第一步: 定义注解处理器, 用来在编译期扫描加入@Route注解的类, 然后做处理。

这也是apt最核心的一步, 新建RouterProcessor 继承自 AbstractProcessor,然后实现process方法。在项目编译期会执行 RouterProcessor的process()方法, 我们便可以在这个方法里处理Route注解了。此时我们需要为RouterProcessor指明它需要处理什么注解, 这里引入一个google开源的自动注册工具AutoService, 如下依赖(也可以手动进行注册, 不过略微麻烦):

```
implementation 'com.google.auto.service:auto-service:1.0-rc2'
```

这个工具可以通过添加注解来为RouterProcessor指定它需要的配置(当然也可以自己手动去配置, 不过会有点麻烦), 如下所示

```
@AutoService(Processor.class)public class RouterProcessor
extends AbstractProcessor { //...}
```

完整的RouterProcessor注解处理器配置如下:

想  
識  
思  
學

```
@AutoService(Processor.class)/** 处理器接收的参数 替代
{@link AbstractProcessor#getSupportedOptions()} 函数
*/@SupportedOptions(Constant.ARGUMENTS_NAME)/** * 指定使用的Java版本 替代 {@link
AbstractProcessor#getSupportedSourceversion()} 函数
*/@SupportedSourceversion(SourceVersion.RELEASE_7)/** * 注册给哪些注解的 替代 {@link
AbstractProcessor#getSupportedAnnotationTypes()} 函数
*/@SupportedAnnotationTypes(Constant.ANNOTATION_TYPE_ROUTE)
public class RouterProcessor extends AbstractProcessor
{
 /** * key:组名 value:类名 */
 private Map<String, String> rootMap = new TreeMap<>(); /**
 * 分组 key:组名 value:对应组的路由信息 */
 private Map<String, List<RouteMeta>> groupMap = new HashMap<>();
 /**
 * 节点工具类 (类、函数、属性都是节点) */
 private Elements elementutils; /**
 * type(类信息)工具类 */
 private Types typeutils; /**
 * 文件生成器 类/资源 */
 private Filer filerutils; /**
 * private
 * String moduleName; /**
 * private Log log; /**
 * @Override
 * public synchronized void init(ProcessingEnvironment
 * processingEnvironment) {
 * super.init(processingEnvironment); /**
 * 获得apt的日志输出
 * log =
 * Log.newLog(processingEnvironment.getMessenger());
 * elementutils = processingEnvironment.getElementUtils();
 * typeutils = processingEnvironment.getTypeUtils();
 * filerutils = processingEnvironment.getFiler();
 * //参数是模块名 为了防止多模块/组件化开发的时候 生成相同的
 * xx$$ROOT$$文件
 * Map<String, String> options =
 * processingEnvironment.getOptions(); /**
 * if
 * (!utils.isEmpty(options)) {
 * moduleName =
 * options.get(Constant.ARGUMENTS_NAME); /**
 * if
 * (utils.isEmpty(moduleName)) {
 * throw new
 * RuntimeException("Not set processor moudleName option
 * !");
 * } /**
 * log.i("init RouterProcessor " +
 * moduleName + " success !");
 * } /**
 * *
 * *
 * @param set 使用了支持处理注解的节点集合
 * @param roundEnvironment 表示当前或是之前的运行环境,可以通过该对象查找找
```

```
到的注解。 * @return true 表示后续处理器不会再处理(已经处理)
 */
@Override public boolean process(Set<?
extends TypeElement> set, RoundEnvironment
roundEnvironment) { if (!utils.isEmpty(set)) {
//被Route注解的节点集合 Set<? extends
Element> rootElements =
roundEnvironment.getElementsAnnotatedWith(Route.class);
if (!utils.isEmpty(rootElements)) {
processorRoute(rootElements); }
return true; } return false; } //...}
```

我们通过@SupportedOptions(Constant.ARGUMENTS\_NAME)拿到每个module的名字，用来生成对应module下存放路由信息的类文件名。在这之前，我们需要在module的gradle下配置如下

```
javaCompileOptions {
annotationProcessorOptions {
[moduleName: project.getName()] arguments =
} }
```

Constant.ARGUMENTS\_NAME便是每个module的名字。

@SupportedAnnotationTypes(Constant.ANNOTATION\_TYPE\_ROUTE)指定了需要处理的注解的路径地址,在此就是Route.class的路径地址。

RouterProcessor中我们实现了init方法，拿到log apt日志输出工具用以输出apt日志信息,并通过以下代码得到上面提到的每个module配置的moduleName

```
//参数是模块名 为了防止多模块/组件化开发的时候 生成相同的
xx$$ROOT$$文件Map<String, String> options =
processingEnvironment.getOptions();if
(!utils.isEmpty(options)) { moduleName =
options.get(Constant.ARGUMENTS_NAME);}if
(utils.isEmpty(moduleName)) { throw new
RuntimeException("Not set processor moudleName option
!");}
```

第二步，在process()方法里开始生成  
EaseRouter\_Route\_moduleName类文件和  
EaseRouter\_Group\_moduleName文件。这里在process()里生成文件用  
javapoet，这是squareup公司开源的一个库，通过调用它的api，可以很  
方便的生成java文件，在含有注解处理器(demo中apt相关的代码实现都  
在easy-compiler module中)的module中引入依赖如下：

```
implementation 'com.squareup:javapoet:1.7.0'
```

好了，我们终于可以生成文件了，在process()方法里有如下代码，

```
if (!utils.isEmpty(set)) { //被Route注解的节点集合
Set<? extends Element> rootElements =
roundEnvironment.getElementsAnnotatedWith(Route.class);
if (!utils.isEmpty(rootElements)) {
processorRoute(rootElements); } return true;}return
false;
```

set就是扫描得到的支持处理注解的节点集合，然后得到rootElements，  
即被@Route注解的节点集合，此时就可以调用  
processorRoute(rootElements)方法去生成文件了。  
processorRoute(rootElements)方法实现如下：

```
private void processorRoute(Set<? extends Element>
rootElements) { //获得Activity这个类的节点信息
TypeElement activity =
elementutils.getTypeElement(Constant.ACTIVITY);
TypeElement service =
elementutils.getTypeElement(Constant.ISERVICE); for
(Element element : rootElements) { RouteMeta
routeMeta; //类信息 TypeMirror typeMirror =
element.asType(); log.i("Route class:" +
typeMirror.toString()); Route route =
element.getAnnotation(Route.class); if
(typeutils.isSubtype(typeMirror, activity.asType())) {
routeMeta = new
RouteMeta(RouteMeta.Type.ACTIVITY, route, element);
} else if (typeutils.isSubtype(typeMirror,
service.asType())) { routeMeta = new
RouteMeta(RouteMeta.Type.ISERVICE, route, element);
} else { throw new RuntimeException("Just
support Activity or IService Route: " + element);
} categories(routeMeta); } TypeElement
iRouteGroup =
elementutils.getTypeElement(Constant.IROUTE_GROUP);
TypeElement iRouteRoot =
elementutils.getTypeElement(Constant.IROUTE_ROOT); //生
成Group记录分组表 generatedGroup(iRouteGroup); //生成
Root类 作用：记录<分组，对应的Group类>
generatedRoot(iRouteRoot, iRouteGroup); }
```

上节中提到过生成的root文件和group文件分别实现了IRouteRoot和IRouteGroup接口，就是通过下面这两行文件代码拿到IRouteGroup和IRouteRoot的字节码信息，然后传入generatedGroup(iRouteGroup)和generatedRoot(iRouteRoot, iRouteGroup)方法，这两个方法内部会通过javapoet api生成java文件，并实现这两个接口。

```
TypeElement iRouteGroup =
elementutils.getTypeElement(Constant.IROUTE_GROUP);TypeEl
ement iRouteRoot =
elementutils.getTypeElement(Constant.IROUTE_ROOT);
```

generatedGroup(iRouteGroup)和generatedRoot(iRouteRoot, iRouteGroup)就是生成上面提到的EaseRouter\_Root\_app和 EaseRouter\_Group\_main等文件的具体实现，代码太多，我粘出一个实现供大家参考，其实生成java文件的思路都是一样的，我们只需要熟悉 javapoet的api如何使用即可。大家可以后续在demo里详细分析，这里我只是讲解核心的实现。

想  
識  
思  
學

```
/** * 生成Root类 作用：记录<分组，对应的Group类> * @param iRouteRoot * @param iRouteGroup */private void generatedRoot(TypeElement iRouteRoot, TypeElement iRouteGroup) { //创建参数类型 Map<String,Class<? extends IRouteGroup>> routes> //wildcard 通配符 ParameterizedTypeName parameterizedTypeName = ParameterizedTypeName.get(ClassName.get(Map.class), ClassName.get(String.class), ParameterizedTypeName.get(ClassName.get(Class.class), WildcardTypeName.subtypeOf(ClassName.get(iRouteGroup)))); //参数 Map<String,Class<? extends IRouteGroup>> routes ParameterSpec parameter = ParameterSpec.builder(parameterizedTypeName, "routes").build(); //函数 public void loadInfo(Map<String,Class<? extends IRouteGroup>> routes) MethodSpec.Builder methodBuilder = MethodSpec.methodBuilder(Constant.METHOD_LOAD_INTO) .addModifiers(Modifier.PUBLIC) .addAnnotation(Override.class) .addParameter(parameter); //函数体 for (Map.Entry<String, String> entry : rootMap.entrySet()) { methodBuilder.addStatement("routes.put($S, $T.class)", entry.getKey(), ClassName.get(Constant.PACKAGE_OF_GENERATE_FILE, entry.getValue())); } //生成$Root$类 String className = Constant.NAME_OF_ROOT + moduleName; TypeSpec typeSpec = TypeSpec.classBuilder(className) .addSuperinterface(ClassName.get(iRouteRoot)) .addModifiers(Modifier.PUBLIC) .addMethod(methodBuilder.build()); try { JavaFile.builder(Constant.PACKAGE_OF_GENERATE_FILE, typeSpec).build().writeTo(filerUtils); log.i("Generated RouteRoot: " + Constant.PACKAGE_OF_GENERATE_FILE + "." + className); } catch (IOException e) { log.e("Failed to generate RouteRoot class: " + e.getMessage()); }}}
```

```
 } catch (IOException e) { e.printStackTrace();
 }
}
```

可以看到，ParameterizedTypeName是创建参数类型的api，ParameterSpec是创建参数的实现，MethodSpec是函数的生成实现等等。最后，当参数、方法、类信息都准备好了之后，调用JavaFileapi生成类文件。JavaFile的builder ()方法传入了PACKAGE\_OF\_GENERATE\_FILE变量，这个就是指定生成的类文件的目录，方便我们在app进程启动的时候去遍历拿到这些类文件。

## 四、框架的设计

好了，如果你看到这里，相信你对路由框架的原理都已经很清楚了。先前有读者问我了一个问题，说为什么初始化的时候不去生成路由的映射文件？在这里，我跟大家再详细说一遍：

首先，我们要明白路由框架作用的两个时期，第一个时期是编译期，第二个时期是app运行的时期。

编译期是在你的项目编译的时候，这个时候还没有开始打包，也就是你没有生成apk呢！路由框架在这个时期根据注解去扫描所有文件，然后生成路由映射文件。这些文件都会统一打包到apk里

app运行时期做的东西也不少，但总而言之都是对映射信息的处理，如执行执行路由跳转等。

### 框架的整体设计

路由框架相对于其他的第三方框架有点特殊，因为它还要在编译期执行操作，所以这里我按照ARouter的思想对框架进行分模块设计，看下图：

我将框架从功能角度分成了三个library，第一个是注解模块，主要放置注解，第二个是compiler模块，里面存放着注解处理器和处理注解的工具类，第三个是路由框架的核心，存放着路由的api等等。其中，easy-compiler 模块和 easy-core 模块都依赖了 easy-annotation 模块，因为它们都需要用到注解。easy-compiler 和 easy-core 没有依赖关系。

为什么我要这么设计，是因为 easy-compiler 模块的作用是通过注解处理器生成路由映射文件，这个注解处理器有些特殊，它是针对模块的，意思就是app模块如果需要用到注解处理器，你就需要在app模块依赖 easy-

compiler, module1模块也需要处理注解，那么就需要module1模块也依赖 easy-compiler，也就是说注解处理器的作用域只针对本module。而 easy-core 模块是框架的核心类，只需要在base模块引入一份就可以，因为所有模块都依赖于base模块。

好了，框架的整体设计已经讲完了，在接下来的文章里，你需要边看代码便阅读，这样才能更好的理解框架的思路。

## 五、框架的初始化

通过前几节的讲解，我们知道了看似很复杂的路由框架，其实原理很简单，我们可以理解为一个map(其实是两个map，一个保存group列表，一个保存group下的路由地址和activityClass关系)保存了路由地址和ActivityClass的映射关系，然后通过map.get("router address")拿到ActivityClass，通过startActivity()调用就好了。但一个框架的设计要考虑的事情远远没有这么简单。下面我们就来分析一下：

要实现这么一个路由框架，首先我们需要在用户使用路由跳转之前把这些路由映射关系拿到手，拿到这些路由关系最好的时机就是应用程序初始化的时候，前面的讲解中我贴过几行代码，是通过apt生成的路由映射关系文件，为了方便大家理解，我把这些文件重新粘贴到下面代码中（这几个类都是单独的文件，在项目编译后会在各个模块的/build/generated/source/apt文件夹下面生成，为了演示方便我只贴出来了app模块下生成的类，其他模块如module1、module2下面的类跟app下面的没有什么区别），在程序启动的时候扫描这些生成的类文件，然后获取到映射关系信息，保存起来。

```
public class EaseRouter_Root_app implements IRouteRoot {
 @Override public void loadInfo(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("main",
 EaseRouter_Group_main.class); routes.put("show",
 EaseRouter_Group_show.class); } } public class
 EaseRouter_Group_main implements IRouteGroup { @Override
 public void loadInfo(Map<String, RouteMeta> atlas) {
 atlas.put("/main/main", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, MainActivity.class, "/main/main", "main"));
 atlas.put("/main/main2", RouteMeta.build(RouteMeta.Type.AC
 TIVITY, Main2Activity.class, "/main/main2", "main"));
 } } public class EaseRouter_Group_show implements
 IRouteGroup { @Override public void
 loadInfo(Map<String, RouteMeta> atlas) {
 atlas.put("/show/info", RouteMeta.build(RouteMeta.Type.ACT
 IVITY, ShowActivity.class, "/show/info", "show")); } }
```

可以看到，这些文件中，实现了IRouteRoot接口的类都是保存了group分组映射信息，实现了IRouteGroup接口的类都保存了单个分组下的路由映射信息。只要我们得到实现IRouteRoot接口的所有类文件，便能通过循环调用它的loadInfo()方法得到所有实现IRouteGroup接口的类，而所有实现IRouteGroup接口的类里面保存了项目的所有路由信息。

IRouteGroup的loadInfo()方法，通过传入一个map，便会将这个分组里的映射信息存入map里。可以看到map里的value是“RouteMeta.build(RouteMeta.Type.ACTIVITY,ShowActivity.class,”/show/info”, “show”)”，RouteMeta.build()会返回RouteMeta，RouteMeta里面便保存着ActivityClass的所有信息。那么我们这个框架，就有了第一个功能需求，便是在app进程启动的时候进行框架的初始化(或者在你开始用路由跳转之前进行初始化都可以)，在初始化中拿到映射关系信息，保存在map里，以便程序运行中可以快速找到路由映射信息实现跳转。下面看具体的初始化代码。

注：这里我们只讲解大体的思路，不会细致到讲解每一个方法每一行代码的具体作用，跟着我的思路你会明白框架设计的具体细节，每一步要实现的功能是什么，但是精确到方法和每一行代码的具体含义你还需要仔细研读demo。

```
public class MyApplication extends Application {
 @Override public void onCreate() {
 super.onCreate(); EasyRouter.init(this);
 }
}

public class EasyRouter {
 private static final String TAG = "EasyRouter";
 private static final String ROUTE_ROOT_PAKCAGE = "com.xsm.easrouter.routes";
 private static final String SDK_NAME = "EaseRouter";
 private static final String SEPARATOR = "_";
 private static final String SUFFIX_ROOT = "Root";
 private static EasyRouter sInstance;
 private static Application mContext;
 private Handler mHandler;
 private EasyRouter() {
 mHandler = new
Handler(Looper.getMainLooper());
 }
 public static EasyRouter getInstance() {
 if (sInstance == null)
 {
 synchronized (EasyRouter.class) {
 if (sInstance == null) {
 sInstance = new EasyRouter();
 }
 }
 }
 return sInstance;
 }
 public static void init(Application application) {
 mContext = application;
 try {
 loadInfo();
 } catch (Exception e) {
 e.printStackTrace();
 Log.e(TAG, "初始化失败!", e);
 }
 }
}
```

可以看到，init()方法中调用了loadInfo()方法，而这个loadInfo()便是我们初始化的核心。

```
private static void loadInfo() throws
PackageManager.NameNotFoundException,
InterruptedException, ClassNotFoundException,
NoSuchMethodException, IllegalAccessException,
InvocationTargetException, InstantiationException { //
获得所有 apt生成的路由类的全类名 (路由表) Set<String>
routerMap = classutils.getFileNameByPackageName(mContext,
ROUTE_ROOT_PAKCAGE); for (String className :
routerMap) { if
(className.startsWith(ROUTE_ROOT_PAKCAGE + "." + SDK_NAME
+ SEPARATOR + SUFFIX_ROOT)) { //root中注册的是分
组信息 将分组信息加入仓库中 ((IRouteRoot)
Class.forName(className).getConstructor().newInstance()).
loadInto(warehouse.groupsIndex); } } for
(Map.Entry<String, Class<? extends IRouteGroup>>
stringClassEntry : warehouse.groupsIndex.entrySet()) {
 Log.d(TAG, "Root映射表[" + stringClassEntry.getKey()
+ " : " + stringClassEntry.getValue() + "]"); }}}
```

我们首先通过ClassUtils.getFileNameByPackageName(mContext, ROUTE\_ROOT\_PAKCAGE)得到apt生成的所有实现IRouteRoot接口的类文件集合，通过上面的讲解我们知道，拿到这些类文件便可以得到所有的routerAddress---activityClass映射关系。

这个ClassUtils.getFileNameByPackageName()方法就是具体的实现了，下面我们看具体的代码：

想  
識  
思  
學

```
/** * 得到路由表的类名 * @param context *
@param packageName * @return * @throws
PackageManager.NameNotFoundException * @throws
InterruptedException */ public static Set<String>
getFileNameByPackageName(Application context, final
String packageName) throws
PackageManager.NameNotFoundException,
InterruptedException { final Set<String>
classNames = new HashSet<>(); List<String> paths =
getSourcePaths(context); //使用同步计数器判断均处理完成
 final CountDownLatch countDownLatch = new
CountDownLatch(paths.size()); ThreadPoolExecutor
threadPoolExecutor =
DefaultPoolExecutor.newDefaultPoolExecutor(paths.size());
 for (final String path : paths) {
threadPoolExecutor.execute(new Runnable() {
@Override public void run() {
DexFile dexFile = null; try {
//加载 apk中的dex 并遍历 获得所有包名为
{packageName} 的类 dexFile = new
DexFile(path); Enumeration<String>
dexEntries = dexFile.entries();
while (dexEntries.hasMoreElements()) {
String className = dexEntries.nextElement();
if (!TextUtils.isEmpty(className)
&& className.startsWith(packageName)) {
classNames.add(className);
}
}
}
catch (IOException e) {
e.printStackTrace(); } finally {
if (null != dexFile) {
try {
dexFile.close(); } catch
(IOException e) {
e.printStackTrace(); }
}
}
//释放一个
countDownLatch.countDown();
}
}); }
}
//
```

```
等待执行完成 countDownLatch.await(); return
classNames; }
```

这个方法会通过开启子线程，去扫描apk中所有的dex，遍历找到所有包名为packageName的类名，然后将类名再保存到classNames集合里。List paths = getSourcePaths(context)这句代码会获得所有的apk文件(instant run会产生很多split apk),这个方法的具体实现大家看demo即可，不再阐述。这里用到了CountDownLatch类，会分path一个文件一个文件的检索，等到所有的类文件都找到后便会返回这个Set集合。所以我们可以知道，初始化时找到这些类文件会有一定的耗时，所以ARouter这里会有一些优化，只会遍历找一次类文件，找到之后就会保存起来，下次app进程启动会检索是否有保存这些文件，如果有就会直接调用保存后的数据去初始化。

## 六、实现路由跳转

通过上节的介绍，我们知道在初始化的时候已经拿到了所有的路由信息，那么实现跳转便好做了。

```
@Route(path = "/main/main")public class MainActivity
extends AppCompatActivity { public void
startModule1MainActivity(View view) {
EasyRouter.getInstance().build("/module1/module1main").n
avigation(); }}
```

在build的时候，传入要跳转的路由地址，build()方法会返回一个Postcard对象，我们称之为跳卡。然后调用Postcard的navigation()方法完成跳转。用过ARouter的对这个跳卡都应该很熟悉吧！Postcard里面保存着跳转的信息。下面我把Postcard类的代码实现粘下来：

想  
識  
思  
學

```
public class Postcard extends RouteMeta { private
Bundle mBundle; private int flags = -1; //新版风格
private Bundle optionsCompat; //老版 private int
enterAnim; private int exitAnim; //服务 private
IService service; public Postcard(String path, String
group) { this(path, group, null); } public
Postcard(String path, String group, Bundle bundle) {
 setPath(path); setGroup(group);
 this.mBundle = (null == bundle ? new Bundle() : bundle);
} public Bundle getExtras() {return mBundle;}
public int getEnterAnim() {return enterAnim;} public
int getExitAnim() {return exitAnim;} public IService
getService() { return service; } public void
setService(IService service) { this.service =
service; } /** * Intent.FLAG_ACTIVITY** *
@param flag * @return */ public Postcard
withFlags(int flag) { this.flags = flag;
return this; } public int getFlags() {
return flags; } /** * 跳转动画 * * @param
enterAnim * @param exitAnim * @return */
public Postcard withTransition(int enterAnim, int
exitAnim) { this.enterAnim = enterAnim;
this.exitAnim = exitAnim; return this; } /**
 * 转场动画 * * @param compat * @return */
public Postcard
withOptionsCompat(ActivityOptionsCompat compat) {
if (null != compat) { this.optionsCompat =
compat.toBundle(); } return this; }
public Postcard withString(@Nullable String key,
@Nullable String value) { mBundle.putString(key,
value); return this; } public Postcard
withBoolean(@Nullable String key, boolean value) {
mBundle.putBoolean(key, value); return this; }
 public Postcard withshort(@Nullable String key, short
value) { mBundle.putShort(key, value);
return this; } public Postcard withInt(@Nullable
String key, int value) { mBundle.putInt(key,
value); return this; } public Postcard
```

```
withLong(@Nullable String key, long value) {
 mBundle.putLong(key, value); return this; }
public Postcard withDouble(@Nullable String key, double
value) { mBundle.putDouble(key, value);
return this; } public Postcard withByte(@Nullable
String key, byte value) { mBundle.putByte(key,
value); return this; } public Postcard
withChar(@Nullable String key, char value) {
mBundle.putChar(key, value); return this; }
public Postcard withFloat(@Nullable String key, float
value) { mBundle.putFloat(key, value);
return this; } public Postcard
withParcelable(@Nullable String key, @Nullable Parcelable
value) { mBundle.putParcelable(key, value);
return this; } public Postcard
withStringArray(@Nullable String key, @Nullable String[]
value) { mBundle.putStringArray(key, value);
return this; } public Postcard
withBooleanArray(@Nullable String key, boolean[] value) {
 mBundle.putBooleanArray(key, value); return
this; } public Postcard withShortArray(@Nullable
String key, short[] value) {
mBundle.putShortArray(key, value); return this;
} public Postcard withIntArray(@Nullable String key,
int[] value) { mBundle.putIntArray(key, value);
return this; } public Postcard
withLongArray(@Nullable String key, long[] value) {
mBundle.putLongArray(key, value); return this;
} public Postcard withDoubleArray(@Nullable String
key, double[] value) { mBundle.putDoubleArray(key,
value); return this; } public Postcard
withByteArray(@Nullable String key, byte[] value) {
mBundle.putByteArray(key, value); return this;
} public Postcard withCharArray(@Nullable String key,
char[] value) { mBundle.putCharArray(key, value);
return this; } public Postcard
withFloatArray(@Nullable String key, float[] value) {
mBundle.putFloatArray(key, value); return this;
```

```
 } public Postcard withParcelableArray(@Nullable
String key, @Nullable Parcelable[] value) {
mBundle.putParcelableArray(key, value); return
this; } public Postcard
withParcelableArrayList(@Nullable String key, @Nullable
ArrayList<? extends Parcelable> value) {
mBundle.putParcelableArrayList(key, value); return
this; } public Postcard
withIntegerArrayList(@Nullable String key, @Nullable
ArrayList<Integer> value) {
mBundle.putIntArrayList(key, value); return
this; } public Postcard
withStringArrayList(@Nullable String key, @Nullable
ArrayList<String> value) {
mBundle.putStringArrayList(key, value); return
this; } public Bundle getOptionsBundle() {
return optionsCompat; } public Object navigation()
{
 return
EasyRouter.getInstance().navigation(null, this, -1,
null); } public Object navigation(Context context)
{
 return
EasyRouter.getInstance().navigation(context, this, -1,
null); } public Object navigation(Context context,
NavigationCallback callback) {
 return
EasyRouter.getInstance().navigation(context, this, -1,
callback); } public Object navigation(Context
context, int requestCode) {
 return
EasyRouter.getInstance().navigation(context, this,
requestCode, null); } public Object
navigation(Context context, int requestCode,
NavigationCallback callback) {
 return
EasyRouter.getInstance().navigation(context, this,
requestCode, callback); }}
```

如果你是一个Android开发，Postcard类里面的东西就不用我再给你介绍了吧！（哈哈）我相信你一看就明白了。我们只介绍一个方法navigation()，他有好几个重载方法，方法里面会调用EasyRouter类的navigation()方法。EaseRouter的navigation()方法，就是跳转的核心了。

下面请看：

数学课业

想  
識  
思  
學

```
protected Object navigation(Context context, final
Postcard postcard, final int requestCode, final
NavigationCallback callback) { try {
prepareCard(postcard); }catch (NoRouteNotFoundException
e) { e.printStackTrace(); //没找到 if
(null != callback) {
callback.onLost(postcard); } return null;
} if (null != callback) {
callback.onFound(postcard); } switch
(postcard.getType()) { case ACTIVITY:
final Context currentContext = null == context ? mContext
: context; final Intent intent = new
Intent(currentContext, postcard.getDestination());
 intent.putExtras(postcard.getExtras());
int flags = postcard.getFlags(); if (-1 !=
flags) { intent.setFlags(flags);
} else if (!(currentContext instanceof Activity)) {
 intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
} mHandler.post(new Runnable() {
@Override
public void run() {
//可能需要返回码
if
(requestCode > 0) {
ActivityCompat.startActivityForResult((Activity)
currentContext, intent,
requestCode, postcard.getOptionsBundle());
} else {
ActivityCompat.startActivity(currentContext, intent,
postcard
.getOptionsBundle()); }
if ((0 != postcard.getEnterAnim() || 0 !=
postcard.getExitAnim()) &&
currentContext instanceof Activity) {
//老版本
((Activity)
currentContext).overridePendingTransition(postcard
.getEnterAnim()
, postcard.getExitAnim());
}
//跳转完成
if (null != callback) {
```

```
callback.onArrival(postcard);
}
ISERVICE:
default:
}) ; break; case
return postcard.getService();
break; } return null; }
```

这个方法里先去调用了prepareCard(postcard)方法,  
prepareCard(postcard)代码我贴出来,

題解課時

```

private void prepareCard(Postcard card) { RouteMeta
routeMeta = warehouse.routes.get(card.getPath()); if
(null == routeMeta) { Class<? extends IRouteGroup>
groupMeta = warehouse.groupsIndex.get(card.getGroup()); if (null == groupMeta) { throw new
NoRouteNotFoundException("没找到对应路由: 分组=" +
card.getGroup() + " 路径=" + card.getPath()); }
 IRouteGroup iGroupInstance; try {
iGroupInstance =
groupMeta.getConstructor().newInstance(); } catch
(Exception e) { throw new RuntimeException("路
由分组映射表记录失败.", e); }
 iGroupInstance.loadInto(warehouse.routes); //已经准
备过了就可以移除了 (不会一直存在内存中)
 warehouse.groupsIndex.remove(card.getGroup()); //再
次进入 else prepareCard(card); } else {
//类 要跳转的activity 或IService实现类
 card.setDestination(routeMeta.getDestination());
 card.setType(routeMeta.getType()); switch
(routeMeta.getType()) { case ISERVICE:
 Class<?> destination = routeMeta.getDestination();
 IService service =
warehouse.services.get(destination); if
(null == service) { try {
 service = (IService)
destination.getConstructor().newInstance();
 warehouse.services.put(destination, service);
 } catch (Exception e) {
 e.printStackTrace();
 }
 card.setService(service);
 break;
 default:
 break;
 }
}
}

```

注意，Warehouse就是专门用来存放路由映射关系的类，这在ARouter里面也是。这段代码Warehouse.routes.get(card.getPath())通过path拿到对应的RouteMeta，这个RouteMeta里面保存了activityClass等信息。继续往下看，如果判断拿到的RouteMeta是空，说明这个路由地址还没有加

载到map里面(为了效率，这里是用了懒加载)，只有在第一次用到当前路由地址的时候，会去Warehouse.routes里面拿routeMeta，如果拿到的是空，会根据当前路由地址的group拿到对应的分组，通过反射创建实例，然后调用实例的loadInfo方法，把它里面保存的映射信息添加到Warehouse.routes里面，并且再次调用prepareCard(card)，这时再通过Warehouse.routes.get(card.getPath())就可以顺利拿到RouteMeta了。

进入else{}里面，调用了

card.setDestination(routeMeta.getDestination())，这个setDestination就是将RouteMeta里面保存的activityClass放入Postcard里面，下面switch代码块可以先不用看，这是实现ARouter中通过依赖注入实现Provider服务的逻辑，有心研究的同学可以去读一下demo。

好了，prepareCard()方法调用完成后，我们的postcard里面就保存了activityClass，然后switch (postcard.getType()){}会判断postcard的type为ACTIVITY，然后通过ActivityCompat.startActivity启动Activity。到这里，路由跳转的实现已经讲解完毕了。

## 七、为什么需要依赖注入

上节我们讲过了如何实现路由的跳转，接下来要为大家讲解一下如何实现依赖注入。

### 什么是依赖注入

做过后台的同学都知道Spring有一大利器依赖注入(简称DI)，什么是依赖注入呢？

举个例子，假如我们的Presenter层需要用网络的数据或数据库的数据，但presenter本身没有请求网络的功能，那么它要依赖model层去请求网络，所以它就需要持有model层的一个实例，我们可以直接通过new的方式去创建，但是这样会使代码耦合严重，那么我们就可以通过一些解耦的手段去让presenter层持有model的引用，比如通过构造方法注入、set方法注入、配置文件注入、注解注入等等，只要达到我们的目的就是好的。

### 路由框架为何需要依赖注入

大家可能会想到，这与我们路由框架有什么关系？诚然，路由的目的就是要实现跳转，但是你有没有想过，两个Activity之间的跳转肯定免不了要传入一些参数，如果我们在跳转后还要通过intent去获取参数，这样岂不是很麻烦，如果可以自动把参数赋给属性多好啊！

如果这个例子不足以说服你，再想一下，组件化中两个module之间可能有一些功能并不需要跳转页面，如支付模块要获取用户模块的用户id，并不需要跳转页面，那么我们就要持有用户模块含有获取用户id功能的类的引用，如果我们在支付模块创建一个用户模块的功能引用，显然就违背了解耦的规则。

这两个问题显然用依赖注入的方式会更好些，如果你用过ARouter，你会发现ARouter中的服务(IProvider)就是通过依赖注入实现的。

## 用什么样的方式实现依赖注入

在多组件并行开发过程中，因为两个module没有引用关系，所以就不能通过构造方法传入要依赖的类，这个时候怎么办呢？连对方的引用都得不到，如何进行依赖呢？可能你会想到反射，我们先pass这个方案，能不用反射就能做好的前提下，我们最好不要用反射。可能有同学会想到，在基类下沉一个接口功能标准，如上面第二个例子中，我们在base模块定义获取用户id的接口，然后在用户模块实现接口的方法。那么当支付模块需要用到这个功能，就声明这个接口，然后一行注解通过框架为你创建实例，这样用户模块只需要提供功能，并不需要关心谁在用这个功能，这样岂不是大大减小了耦合。

为什么要用注解实现依赖注入，因为我们用了apt啊，那岂不是天生实现依赖注入的利器。如果你去写配置文件或构造方法等等，未免太复杂。最主要的，看着牛逼。

## 八、Activity的属性注入

上节讲了为什么需要依赖注入，这节我们就来实现依赖注入。因为依赖注入包含两大块，其一是activity通过intent传参属性的注入，其二是类似ARouter的IPorvider。其实两者的实现方式类似，但是考虑到文章长度的问题，我们先来讲解第一个。

### 依赖注入实现

先来看一段代码：

```
public class MainActivity { public void
startModule1MainActivity(View view) { //跳转到
Module1MainActivity, 并且传入参数msg
EasyRouter.getInstance().build("/module1/module1main")
 .withString("msg", "从
MainActivity").navigation(); } } @Route(path =
"/module1/module1main") public class Module1MainActivity
extends AppCompatActivity { @Extra String msg;
@Override protected void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_module1_main);
EasyRouter.getInstance().inject(this);
Toast.makeText(this, "msg=" + msg,
Toast.LENGTH_SHORT).show(); }}
```

我们要通过注解处理器实现MainActivity传入的msg参数直接被Module1MainActivity解析，需要两个要素，其一是需要知道哪个类要被注入，很显然，这里是Module1MainActivity，其二是这个类的哪一个属性要被注入，这里就是msg。

这里我就以这个例子来说明，只要我们在编译期间拿到需要依赖注入的类Module1MainActivity和需要注入的属性msg，那么在Module1MainActivity的onCreate方法里，我们调用EasyRouter.getInstance().inject(this)方法把intent中的属性注入到msg里面就可以了。

这个属性msg如何拿到呢？还是用注解处理器，我们在msg属性上加上注解@Extra，然后在编译期拿到所有被Extra注解的属性集。

说起来简单，下面我们看代码：

想  
識  
思  
學

```
@AutoService(Processor.class)@SupportedOptions(Constant.ARGUMENTS_NAME)@SupportedSourceVersion(SourceVersion.RELEASE_7)@SupportedAnnotationTypes({Constant.ANN_TYPE_EXTRA})
public class ExtraProcessor extends AbstractProcessor {
 /** * 节点工具类 (类、函数、属性都是节点) */ private
 Elements elementutils; /** * type(类信息)工具类 */
 private Types typeutils; /** * 类/资源生成器 */
 /** private Filer filerutils; /** * 记录所有需要
 注入的属性 key:类节点 value:需要注入的属性节点集合 */
 private Map<TypeElement, List<Element>> parentAndChild =
 new HashMap<>(); private Log log; @Override
 public synchronized void init(ProcessingEnvironment
 processingEnvironment) {
 super.init(processingEnvironment); //获得apt的日志输出
 log =
 Log.newLog(processingEnvironment.getMessager());
 elementutils = processingEnv.getElementutils();
 typeutils = processingEnvironment.getTypeutils();
 filerutils = processingEnv.getFiler(); } @Override
 public boolean process(Set<? extends TypeElement> set,
 RoundEnvironment roundEnvironment) { if
 (!utils.isEmpty(set)) { Set<? extends Element>
 elements =
 roundEnvironment.getElementsAnnotatedwith(Extra.class);
 if (!utils.isEmpty(elements)) {
 try { categories(elements);
 generateAutowired(); } catch
 (IOException e) { e.printStackTrace();
 } return true;
 }
 } return false;
 } private void
 generateAutowired() throws IOException {
 TypeMirror typeActivity =
 elementutils.getTypeElement(Constant.ACTIVITY).asType();
 TypeElement iExtra =
 elementutils.getTypeElement(Constant.IEXTRA); if
 (!utils.isEmpty(parentAndChild)) { // 参数
 Object target ParameterSpec objectParamSpec =
 ParameterSpec.builder(TypeName.OBJECT, "target").build();
 }
```

```
for (Map.Entry<TypeElement, List<Element>>
entry : parentAndChild.entrySet()) {
TypeElement rawClassElement = entry.getKey();
if (!typeutils.isSubtype(rawClassElement.asType(),
typeActivity)) { throw new
RuntimeException("just support activity filed: " +
rawClassElement); } //封装的
函数生成类 LoadExtraBuilder loadExtra = new
LoadExtraBuilder(objectParamSpec);
loadExtra.setElementutils(elementutils);
loadExtra.setTypeutils(typeutils);
ClassName className = className.get(rawClassElement);
loadExtra.injectTarget(className);
//遍历属性 for (int i = 0; i <
entry.getValue().size(); i++) {
Element element = entry.getValue().get(i);
loadExtra.buildStatement(element);
// 生成java类名 String
extraClassName = rawClassElement.getSimpleName() +
Constant.NAME_OF_EXTRA; // 生成
xx$$Autowired
JavaFile.builder(className.packageName(),
TypeSpec.classBuilder(extraClassName)
.addSuperinterface(className.get(iExtra))
.addModifiers(PUBLIC).addMethod(loadExtra.build()).build()
).build().writeTo(fileutils);
log.i("Generated Extra: " +
className.packageName() + "." + extraClassName);
} } /** * 记录需要生成的类与属性 */
* @param elements * @throws
IllegalAccessException */ private void
categories(Set<? extends Element> elements) { for
(Element element : elements) { //获得父节点 (类)
TypeElement enclosingElement = (TypeElement)
element.getEnclosingElement(); if
(parentAndChild.containsKey(enclosingElement)) {
parentAndChild.get(enclosingElement).add(element); }}
```

```
 } else {
 new ArrayList<>(); childs =
 childs.add(element);
 parentAndChild.put(enclosingElement,
childs); } }}}
```

ExtraProcessor是处理Activity参数注入的注解处理器，关于注解处理器前面已经讲解过，我们就不在说了。直接看process()方法，我们拿到所有被Extra注解的属性集合，然后调用categories(elements)方法，categories()方法里会将属性所在类和属性集合保存起来，接着调用generateAutoWired()方法。generateAutoWired()方法的作用就是将要依赖的类和依赖的属性信息以java文件的形式存起来，这些利用javapoet生成Java类的api我们以前已经讲过了，这里为了保持代码的美观把这些生成过程封装了起来，还是以上面的例子，将会生成关系文件如下：

```
public class Module1MainActivity_Extra implements IExtra
{ @Override public void loadExtra(Object target) {
Module1MainActivity t = (Module1MainActivity)target;
t.msg = t.getIntent().getStringExtra("msg"); }}
```

我们生成的类文件以Module1MainActivity开头，具有一定的规则，并且实现了IExtra接口的loadExtra(Object target)方法，只要有了这个方法，我们便可以在程序运行期间让Module1MainActivity调用此方法，将intent里面的属性注入到msg属性上。

## api设计

上面我说过，依赖注入需要两个要素，其一是需要知道哪个类要被注入，其二是这个类的哪一个属性要被注入。

我们通过注解处理器解析@Extra注解并生成文件，这些文件记录了需要被注入的类和属性，那么如何注入以及在什么时期注入呢？在什么时期注入是根据具体业务而定的，那么框架只需要提供一个注入的方法，将在何时注入交给需要被注入的类就可以了。

在Module1MainActivity的onCreate方法里，我调用了一下代码：

```
EasyRouter.getInstance().inject(this);
```

这个就是注入的方法，注入的时期Activity的onCreate时期。下面看inject()方法：

```
public void inject(Activity instance) {
 ExtraManager.getInstance().loadExtras(instance);}
public class ExtraManager {
 public void loadExtras(Activity instance) {
 //查找对应activity的缓存
 String className = instance.getClass().getName();
 IExtra iExtra = classCache.get(className);
 try {
 if (null == iExtra) {
 iExtra = (IExtra)
 Class.forName(instance.getClass().getName() +
 SUFFIX_AUTOWIRED).getConstructor().newInstance();
 }
 iExtra.loadExtra(instance);
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

首先我们拿到类名，然后从classCache里取出生成Module1MainActivity映射关系的类Module1MainActivity\_Extra，这里classCache就是一个缓存，为的是如果打开过Module1MainActivity后再次打开就不需要重新创建Module1MainActivity\_Extra去拿映射信息。

接着看，我们通过IExtra iExtra = classCache.get(className)拿到iExtra，这里的iExtra就是Module1MainActivity\_Extra，如果拿到的是空，就去通过反射创建Module1MainActivity\_Extra实例，然后调用loadExtra(instance)方法将msg从intent里面拿出来赋给msg属性。

好了，到这里，Activity的属性注入就讲完了，其实很简单，如果有读者还是不理解，我建议你将源码中的每一个流程再详细看一遍。

### 20.3.5 Arouter核心思路和源码详解

#### 前言

阅读本文之前，建议读者：

- 对Arouter的使用有一定的了解。
- 对Apt技术有所了解。

Arouter是一款Alibaba出品的优秀的路由框架，本文不对其进行全面的分析，只对其最重要的功能进行源码以及思路分析，至于其拦截器，降级，ioc等功能感兴趣的同学请自行阅读源码，强烈推荐阅读云栖社区的[官方介绍](#)。

对于一个框架的学习和讲解，我个人喜欢先将其最核心的思路用简单一两句话总结出来：ARouter通过Apt技术，生成保存路径(路由path)和被注解(@Router)的组件类的映射关系的类，利用这些保存了映射关系的类，ARouter根据用户的请求postcard（明信片）寻找到要跳转的目标地址(class)，使用Intent跳转。

原理很简单，可以看出来，该框架的核心是利用apt生成的映射关系，这里要用到Apt技术，读者可以自行搜索了解一下。

## 分析

我们先看最简单的代码的使用：

首先需要在需要跳转的组件添加注解

```
@Route(path = "/main/homepage")public class HomeActivity
extends BaseActivity { onCreate() }
```

然后在需要跳转的时候调用

```
ARouter.getInstance().build("main/hello").navigation;
```

这里的路径“main/hello”是用户唯一配置的东西，我们需要通过这个path找到对应的Activity。最简单的思路就是通过APT技术，寻找到所有带有注解@Router的组件，将其注解值path和对应的Activity保存到一个map里，比如像下面这样：

```
class RouterMap { public Map getAllRoutes {
 Map map = new HashMap<String,Class<?>>;
 map.put("/main/homepage",HomeActivity.class);
 map.put("/main/setting",SettingActivity.class);
 map.put("/login/register",LoginRegisterActivity.class);
 return map; }}
```

然后在工程代码中将这个map加载到内存中，需要的时候直接get(path)就可以了，这种方案似乎能解决我们的问题。

## 发现问题

上面的思路确实能够实现路由功能，但是这么做会存在一个较大的问题：对于一个大型项目，组件数量会很多，可能会有一两百或者更多，把这么多组件都放到这个Map里，显然会对内存造成很大的压力，因此，Arouter作为一款阿里出品的优秀框架，显然是要解决这个问题的。

这里建议读者自行思考一下，如何解决一次性加载所有映射关系带来的内存损耗问题，我在思考这个问题的时候首先想到的是“懒加载”，但是仅仅懒加载是不够的，因为懒加载后如果还是一次性把所有映射关系加载进来，内存损耗还是一样大的。因此，再深入思考一下，可能还能想出解决一个思路：分段懒加载，思路有了，如何实现呢？这里还是建议大家在阅读下面的内容之前思考一下，或许你能想到一套不同于Arouter的方案来哦。

Arouter采用的方法就是“分组+按需加载”，分组还带来的另一个好处是便于管理，下面我们来看一下实现原理。

### 解决步骤一：分组

首先看如何分组的，Arouter在一层map之外，增加了一层map，我们看WareHouse这个类，里面有两个静态Map：

```
static Map<String, Class<? extends IRouteGroup>>
groupsIndex = new HashMap<>(); static Map<String,
RouteMeta> routes = new HashMap<>();
```

- groupsIndex 保存了group名字到IRouteGroup类的映射，这一层映射就是Arouter新增的一层映射关系。
- routes 保存了路径path到RouteMeta的映射，其中，RouteMeta是目标地址的包装。这一层映射关系跟我们自己方案里的map是一致的，我们路径跳转也是要用这一层来映射。

这里出现了两个我们不认识的类，IRouteGroup和RouteMeta，后者很简单，就是对跳转目标的封装，我们后续称其为“目标”，其内包含了目标组件的信息，比如Activity的Class。那IRouteGroup是个什么东西？

```
public interface IRouteGroup { /** * Fill the
atlas with routes in group. */ void
loadInto(Map<String, RouteMeta> atlas);}
```

一个接口，只有一个方法loadInto，都有谁实现了这个接口呢？我拿我手上的一个项目为例，Arouter通过apt生成了下面几个类：



这几个类都以Arouter\$\$Group开头，我们随便拿一个看看：

```
public class ARouter$$Group$$main implements IRouteGroup
{ @Override public void loadInto(Map<String, RouteMeta>
atlas) { atlas.put("/main/fa/leakscan",
RouteMeta.build(RouteType.ACTIVITY,
MainFaLeakScanActivity.class, "/main/fa/leakscan",
"main", {}, -1, 1)); atlas.put("/main/login",
RouteMeta.build(RouteType.ACTIVITY, LoginActivity.class,
"/main/login", "main", null, -1, -2147483648));
atlas.put("/main/register",
RouteMeta.build(RouteType.ACTIVITY,
RegPhoneActivity.class, "/main/register", "main", null,
-1, -2147483648)); }}
```

我们看到，他实现了loadInto方法，在这个方法中，它往这个HashMap中填充了好多数据，填充的是什么呢？填充的是路径path和它对应的目标RouteMeta，也就是我们最终需要的那层映射关系。而且，我们还能观察到：这个类下面所有的路由path都有一个共同点，即全是“/main”开头的，也就是说，这个类加载的映射关系，都是在一个组内的。因此我们总结出：

**Arouter通过apt技术，为每个组生成了一个以Arouter\$\$Group开头的类，这个类负责向atlas这个Hashmap中填充组内路由数据。**

IRouteGroup正如其名字，它就是一个能装载该组路由映射数据的类，其实有点像个工具类，为了方便后续讲解，我们姑且称上面这样一个实现了IRouteGroup的类叫做“组加载器”，本质是一个类。上图中的类是一个组加载器，其他所有以Arouter\$\$Group开头的类都是一个“组加载器”。回到之前的主线，Warehoust中的两个HashMap，其中groupsIndex这个map中保存的是什么呢？我们通过它的调用找到这一行代码(已简化)：

```
for (String className : routerMap) { if
(className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME
+ SEPARATOR + SUFFIX_ROOT)) { ((IRouteRoot)
(Class.forName(className).getConstructor().newInstance())
).loadInto(warehouse.groupsIndex); }}
```

其中 ROUTE\_ROOT\_PAKCAGE + DOT + SDK\_NAME + SEPARATOR + SUFFIX\_ROOT 这行代码是几个静态字符串拼起来的，它等于 com.alibaba.android.arouter.routes.Arouter\$\$Root。另外 routerMap是什么呢？它是一个HashSet：

```
routerMap = classutils.getFileNameByPackageName(mContext,
ROUTE_ROOT_PAKCAGE);
```

这一行代码对它进行了初始化，目的是找到 com.alibaba.android.arouter.routes 这个包名下所有的类，将其类名保存到routerMap中。因此，上面的代码意思就是将 com.alibaba.android.arouter.routes 包下所有名字以 com.alibaba.android.arouter.routes.Arouter\$\$Root 开头的类找出来，通过反射实例化并强转成IRouteRoot，然后调用loadInto方法。这里又出来一个新的接口：IRouteRoot，我们看代码：

```
public interface IRouteRoot { /** * Load routes to
input * @param routes input */ void
loadInto(Map<String, Class<? extends IRouteGroup>>
routes);}
```

跟IRouteGroup长得还挺像，也是loadInto，我们看它的实现。还是以我的项目为例，在apt生成的文件夹下查找：



最底下一行，有个ARouter\$\$Root\$\$app，它符合前面名字规则，我们进去看看：

```
public class ARouter$$Root$$app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("YDY",
 ARouter$$Group$$YDY.class); routes.put("app",
 ARouter$$Group$$app.class); routes.put("main",
 ARouter$$Group$$main.class); routes.put("payment",
 ARouter$$Group$$payment.class); routes.put("wallet",
 ARouter$$Group$$wallet.class); } }
```

这个类实现了IRouteRoot，在loadInto方法中，他将组名和组对应的“组加载器”保存到了routes这个map中。也就是说，这个类将所有的“组加载器”给索引了下来，通过任意一个组名，可以找到对应的“组加载器”，我们再回到前面讲的初始化Arouter时候的方法中：

```
((IRouteRoot)
(Class.forName(className).getConstructor().newInstance())
.loadInto(warehouse.groupsIndex);
```

理解了吧，这个方法的意义就在于将所有的组路由加载类索引到了groupsIndex这个map中。因此我们就明白了：

## WareHouse中的groupsIndex保存的是组名到“组加载器”的映射关系

说句题外话：回过头想想前面用到的两个接口：IRouteGroup和IRouteRoot，它们其实是apt生成的类和我们项目中代码之间沟通的桥梁，熟悉AIDL的同学可能会觉得很熟悉，二者其实是异曲同工的，两个系统进行交互的时候都是通过接口来沟通的。当然，在使用apt生成的类时，我们需要用到反射技术。

总结一下Arouter的分组设计：Arouter在原来path到目标的map外，加了一个新的map，该map保存了组名到“组加载器”的映射关系。其中“组加载器”是一个类，可以加载其组内的path到目标的映射关系。

到此为止，Arouter只是完成了分组工作，但这么做的目的是什么呢？别着急，前面的都只是铺垫，接下来才是这个分组设计发挥作用的地方，我们进入“按需加载”的代码分析：

## 解决步骤二：按需加载

之前说过，Arouter使用的是分组按需加载，分组是为了按需做准备的。我们看Arouter是怎么按需加载的，我们还是从代码的使用入手：

```
Arouter.getInstance().build("main/hello").navigation;
```

在navigation这个方法中，最终会跳转到这里：

```
protected Object navigation(final Context context, final
Postcard postcard, final int requestCode, final
NavigationCallback callback) { try { //请
关注这一行 LogisticsCenter.completion(postcard);
} catch (NoRouteNotFoundException ex) {
Logger.warning(Consts.TAG, ex.getMessage());
....//简化代码 } //调用Intent跳转
return _navigation(context, postcard, requestCode,
callback)
```

最后一行的return语句很简单，就是去调用Intent唤起组件了，我们看前面try中的第一行 `LogisticsCenter.completion(postcard)`，我们进入到这个函数里：

```
//从缓存里取路由信息RouteMeta routeMeta =
Warehouse.routes.get(postcard.getPath()); //如果为空，需要加载该组的路由
if (null == routeMeta) { Class<? extends
IRouteGroup> groupMeta =
Warehouse.groupsIndex.get(postcard.getGroup());
IRouteGroup iGroupInstance =
groupMeta.getConstructor().newInstance();
iGroupInstance.loadInto(Warehouse.routes);
Warehouse.groupsIndex.remove(postcard.getGroup()); } //如果不为空，走后续流程
else {
postcard.setDestination(routeMeta.getDestination());
... }
```

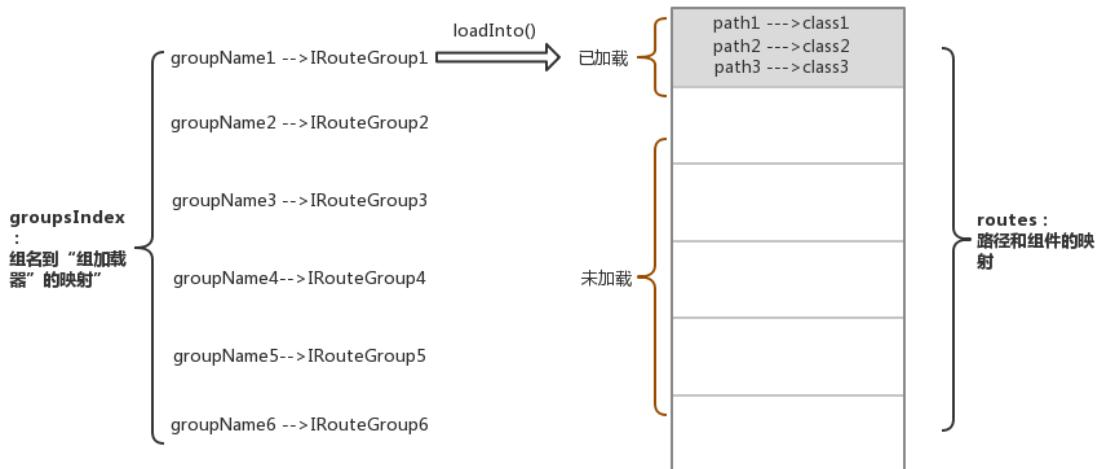
这段代码就是“按需加载”的核心逻辑所在了，我对其进行了简化，分析其逻辑是这样的：

- 首先从Warehouse.routes(前面说了，这里存放的是path到目标的映射)里拿到目标信息，如果找不到，说明这个信息还没加载，需要加载，实际上，刚开始这个routes里面什么都没有。
- 加载流程：首先从Warehouse.groupsIndex里获取“组加载器”，组加载器是一个类，需要通过反射将其实例化，实例化为iGroupInstance，接着调用组加载器的加载方法loadInto，将该组的路由映射关系加载到Warehouse.routes中，加载完成后，routes中就缓存下来当前组的所有路由映射了，因此这个组加载器其实就没用了，为了节省内存，将其从Warehouse.groupsIndex移除。
- 如果之前加载过，则在Warehouse.routes里面是可以找到路有映射关系的，因此直接将目标信息routeMeta传递给postcard，保存在postcard中，这样postcard就知道了最终要去哪个组件了。

到此为止分组按需加载的逻辑就都分析完了，通过这两个步骤，解决了路由映射一次性加载到内存占用内存过大的缺点，这是Arouter这个框架优秀的重要原因之一。当然Arouter还有一些优秀的功能，比如拦截器，依赖注入等，总之，功能全，性能好，使用方便，这些都是Arouter受欢迎的原因，这点值得我们所有开发者去学习。

## 总结

最后结合一张图总结一下Arouter的分组按需加载的逻辑：



图中左侧groupsIndex是“组映射”，右侧routes是“路由映射”。ARouter在初始化的时候，通过反射技术，将所有的“组加载器”索引到groupsIndex这个map中，而此时，右侧的routes还是空的。在用户调用navigation()进行跳转的时候，会根据路径提取组名，由组名根据groupsIndex获取到相应组的“组加载器”，由组加载器加载对应组内的路由信息，此时保存全局“路由目标映射”的routes这个map中就保存了刚才组的所有路由映射关系了。同样，当其他组请求时，其他组也会加载组对应的路由映射，这样就实现了整个App运行时，只有用到的组才会加到内存中，没有去过的组就不会加载到内存中，达到了节省内存的目的。

## 20.4.6 ARouter源码分析——缓存与优化

ARouter 源码分析系列建议从最初开始阅读，全部文章请访问  
<https://github.com/AlexMahao/ARouter>

本篇博客意在记录ARouter中的一些优秀策略。

### 辅助类加载机制

ARouter在实现基本功能时，使用apt在指定包名下生成了一些辅助类。辅助类的查询逻辑如下。

```
if (ARouter.debuggable() ||
 Packageutils.isNewVersion(context)) {
 logger.info(TAG, "Run with debug mode or new install,
 rebuild router map."); // These class
 was generated by arouter-compiler.
 获取com.alibaba.android.arouter.routes的类列表，及Processor
 生成的辅助类
 routerMap =
 Classutils.getFileNameByPackageName(mContext,
 ROUTE_ROOT_PAKCAGE);
 if
 (!routerMap.isEmpty()) { // 保存路由
 表缓存
 context.getSharedPreferences(AROUTER_SP_CACHE_KEY,
 Context.MODE_PRIVATE).edit().putStringSet(AROUTER_SP_KEY_
 MAP, routerMap).apply();
 // 修改新的路由表版本
 Packageutils.updateVersion(context); // Save new
 version name when router map update finishes.
 } else {
 logger.info(TAG, "Load
 router map from cache.");
 routerMap =
 new HashSet<>
 (context.getSharedPreferences(AROUTER_SP_CACHE_KEY,
 Context.MODE_PRIVATE).getStringSet(AROUTER_SP_KEY_MAP,
 new HashSet<String>()));
 }
}
```

判断是否是 debug 或者是否是新版本，其中一个成立，则重新查询并且遍历一下指定包下的所有辅助类。并且将辅助类的全路径保存到 sp 中。

如果不满足条件，则从 sp 中取。

尽可能减少耗时操作调用的次数。

## 路由的懒加载

对于一个大型 app，存在的路由地址数量很大。而 ARouter 的初始化方法都是在 Application 中，那么势必导致 app 加载时间过长。ARouter 通过分组懒加载的形式进行加载。

ARouter 在初始化的时候，只是加载了分组的类，即路由清单的 class，而没有加载详细的路由清单。

```
public class ARouter$$Root$$app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("test",
 ARouter$$Group$$test.class);
 routes.put("yourservicegroupname",
 ARouter$$Group$$yourservicegroupname.class); }}}
```

然后当有路由地址跳转的时候，判断是否能查寻对应地址信息，如果没有，则会根据路由地址确定其分组，然后加载该组的路由清单。

```
public synchronized static void completion(Postcard postcard) { if (null == postcard) { throw new NoRouteNotFoundException(TAG + "No postcard!"); } // 获取路由地址对应的信息 RouteMeta routeMeta = warehouse.routes.get(postcard.getPath()); // 如果为null可能是由于对应组别的路由清单没有加载 if (null == routeMeta) { // Maybe its does't exist, or didn't load. // 查询对应组的路由清单 Class<? extends IRouteGroup> groupMeta = warehouse.groupsIndex.get(postcard.getGroup()); // Load route meta. if (null == groupMeta) { // 如果不存在, 则说明当前路径不存在 throw new NoRouteNotFoundException(TAG + "There is no route match the path [" + postcard.getPath() + "], in group [" + postcard.getGroup() + "]"); } else { // Load route and cache it into memory, then delete from metas. try { if (ARouter.debuggable()) { logger.debug(TAG, String.format(Locale.getDefault(), "The group [%s] starts loading, trigger by [%s]", postcard.getGroup(), postcard.getPath())); } // 加载分组的路由清单 IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstance(); iGroupInstance.loadInto(warehouse.routes); // 已加载的路由清单, 将其从根节点移除 warehouse.groupsIndex.remove(postcard.getGroup()); if (ARouter.debuggable()) { logger.debug(TAG, String.format(Locale.getDefault(), "The group [%s] has already been loaded, trigger by [%s]", postcard.getGroup(), postcard.getPath())); } } catch (Exception e) { throw new HandlerException(TAG + "Fatal exception when loading group meta. [" + e.getMessage() + "]"); } // 重新查询 completion(postcard); // Reload } } }
```

## 20.4.7 我所理解的Android组件化之通信机制

### 1. 概述

之前写过一篇关于Android组件化的文章，《Android组件化框架设计与实践》<https://www.jianshu.com/p/1c5afe686d75>，之前没看过的小伙伴可以先点击阅读。

那篇文章是从实战中进行总结得来，是公司的一个真实项目进行组件化架构改造，粒度会分的更粗些，是对整体架构实践进行相应的总结，里面说了要打造一个组件化框架的话，需要从以下7个方面入手：

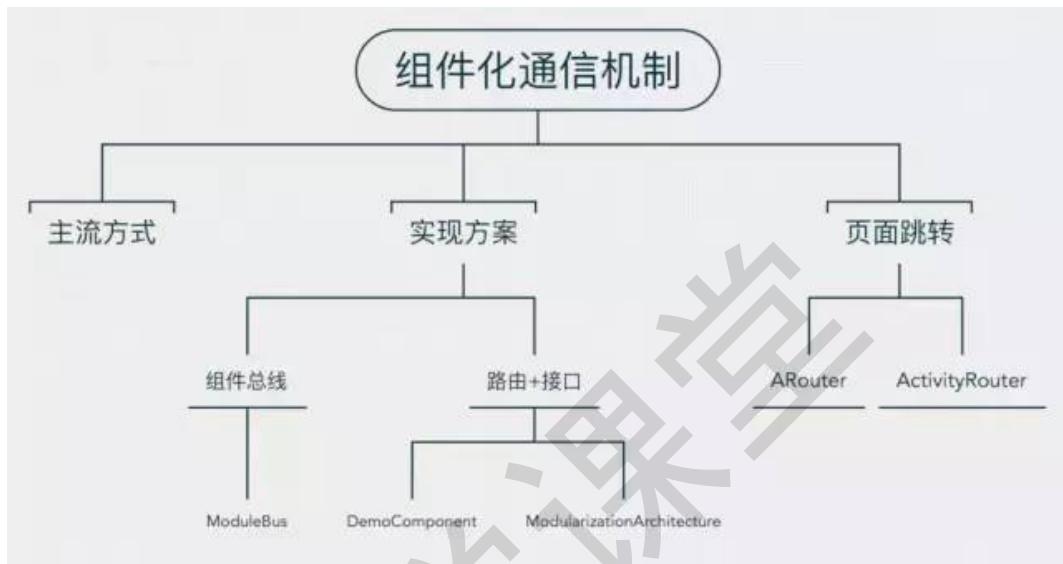
1. 代码解耦。如何将一个庞大的工程分成有机的整体？
2. 组件单独运行。因为每个组件都是高度内聚的，是一个完整的整体，如何让其单独运行和调试？
3. 组件间通信。由于每个组件具体实现细节都互相不了解，但每个组件都需要给其他调用方提供服务，那么主项目与组件、组件与组件之间如何通信就变成关键？
4. UI 跳转。UI 跳转指的是特殊的数据传递，跟组件间通信区别有什么不同？
5. 组件生命周期。这里的生命周期指的是组件在应用中存在的时间，组件是否可以做到按需、动态使用、因此就会涉及到组件加载、卸载等管理问题。
6. 集成调试。在开发阶段如何做到按需编译组件？一次调试中可能有一两个组件参与集成，这样编译时间就会大大降低，提高开发效率。
7. 代码隔离。组件之间的交互如果还是直接引用的话，那么组件之间根本没有做到解耦，如何从根本上避免组件之间的直接引用，也就是如何从根本上杜绝耦合的产生？

今天则会从更小粒度入手，主要讲讲在组件化架构下组件与组件之间通信机制是如何、包括所谓的UI跳转，其实也是组件化通信，只不过它稍微特殊点，单独抽取出来而已。

学习知识的过程很常见的一个思路就是从整体概况入手，首先对整体有个粗略的印象，然后再深入细节，抽丝剥茧般去挖掘其中的内在原理，一个点一个不断去突破，这样就能建立起自己整个知识树，所以今天我们就从通信机制这个点入手，看看其中内在玄机有哪些。

## 2.思维导图

同样，在每写一篇文章之前，放个思维导图，这样做好处对于想写的内容有很好的梳理，逻辑和结构上显得清晰点。



## 3.主流方式

总所周知，Android提供了很多不同的信息的传递方式，比如在四大组件中本地广播、进程间的AIDL、匿名间的内存共享、Intent Bundle传递等等，那么在这么多传递方式，哪种类型是比较适合组件与组件直接的传递呢。

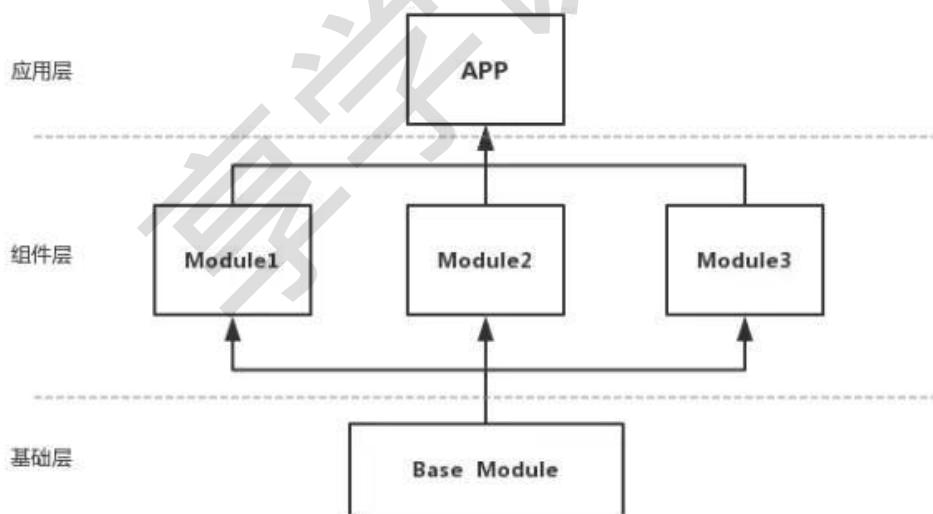
- 本地广播，也就是LoacalBroadcastRecevier。更多是用在同一个应用内的不同系统规定的组件进行通信，好处在于：发送的广播只会在自己的APP内传播，不会泄漏给其他的APP，其他APP无法向自己的APP发送广播，不用被其他APP干扰。本地广播好比对讲通信，成本低，效率高，但有个缺点就是两者通信机制全部委托与系统负责，我们无法干预传输途中的任何步骤，不可控制，一般在组件化通信过程中采用比例不高。
- 进程间的AIDL。这个粒度在于进程，而我们组件化通信过程往往是在线程中，况且AIDL通信也是属于系统级通信，底层以Binder机制，虽

说Android提供模板供我们实现，但往往使用者不好理解，交互比较复杂，往往也不适用于组件化通信过程中。

- 匿名的内存共享。比如用Sharedpreferences，在处于多线程场景下，往往会线程不安全，这种更多是存储一些变化很少的信息，比如说组件里的配置信息等等。
- Intent Bundle传递。包括显性和隐性传递，显性传递需要明确包名路径，组件与组件往往是需要互相依赖，这背离组件化中SOP（关注点分离原则），如果走隐性的话，不仅包名路径不能重复，需要定义一套规则，只有一个包名路径出错，排查起来也稍显麻烦，这种方式往往在组件间内部传递会比较合适，组件外与其他组件打交道则使用场景不多。

说了这么多，那组件化通信什么机制比较适合呢？既然组件层中的模块是相互独立的，它们之间并不存在任何依赖。没有依赖就无法产生关系，没有关系，就无法传递消息，那要如何才能完成这种交流？

目前主流做法之一就是引入第三者，比如图中的Base Module。



## 基础组件化架构

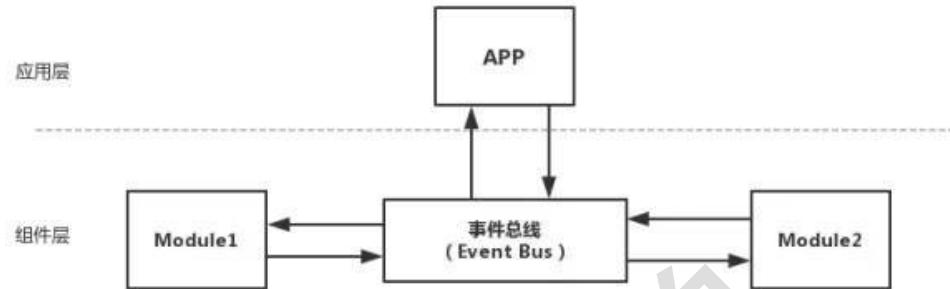
组件层的模块都依赖于基础层，从而产生第三者联系，这种第三者联系最终会编译在APP Module中，那时将不会有这种隔阂，那么其中的Base Module就是跨越组件化层级的关键，也是模块间信息交流的基础。比较有代表性的组件化开源框架有得到DDComponentForAndroid、阿里Arouter、聚美Router等等。

<https://github.com/luojilab/DDComponentForAndroid>

<https://github.com/alibaba/Arouter>

<https://github.com/JumeiRdGroup/Router>

除了这种以通过引入第三者方式，还有一种解决方式是以事件总线方式，但这种方式目前开源的框架中使用比例不高，如图：



## 事件总线

事件总线通过记录对象，使用监听者模式来通知对象各种事件，比如在现实生活中，我们要去找房子，一般都去看小区的公告栏，因为那边会经常发布一些出租信息，我们去查看的过程中就形成了订阅的关系，只不过这种是被动去订阅，因为只有自己需要找房子了才去看，平时一般不会去看。

小区中的公告栏可以想象成一个事件总线发布点，监听者则是哪些想要找房子的人，当有房东在公告栏上贴上出租房信息时，如果公告栏有订阅信息功能，比如引入门卫保安，已经把之前来这个公告栏要查看的找房子人——进行电话登记，那么一旦有新出租消息产生，则门卫会把这条消息一一进行短信群发，那么找房子人则会收到这条消息进行后续的操作，是马上过来看，还是延迟过来，则根据自己的实际情况进行处理。

在目前开源库中，有EventBus、RxBus就是采用这种发布/订阅模式，优点是简化了Android组件之间的通信方式，实现解耦，让业务代码更加简洁，可以动态设置事件处理线程和优先级，缺点则是每个事件需要维护一个事件类，造成事件类太多，无形中加大了维护成本。那么在组件化开源框架中有ModuleBus、CC 等等。

<https://github.com/cangwang/ModuleBus>

<https://github.com/luckybilly/CC>

这两者模式更详细的对比，可以查看这篇文章多个维度对比一些有代表性的开源android组件化开发方案。

<https://github.com/luckybilly/AndroidComponentizeLibs>

#### 4. 实现方案

事件总线，又可以叫做组件总线，路由+接口，则相对好理解点，今天从阅读它们框架源码，我们来对比这两种实现方案的不同之处。

#### 组件总线

这边选取的是ModuleBus框架，这个方案特别之处在于其借鉴了EventBus的思想，组件的注册/注销和组件调用的事件发送都跟EventBus类似，能够传递一些基础类型的数据，而并不需要在Base Moudel中添加额外的类。

所以不会影响Base模块的架构，但是无法动态移除信息接收端的代码，而自定义的事件信息类型还是需要添加到Base Module中才能让其他功能模块索引。

其中的核心代码是在与ModuleBus类，其内部维护了两个ArrayMap键对值列表，如下

```
/* ObjectmethodClass */ StringmethodName; /*
MethodInfo method info */ private static ArrayMap<
Object, ArrayMap< String, MethodInfo>> moduleEventMethods =
new ArrayMap<>(); /* classIBaseClient. class */
StringmethodName /*
ObjectmethodClass */ private static ArrayMap< Class< ?>,
ArrayMap< String, ArrayList< Object>>>
moduleMethodClient = new ArrayMap<>();
```

在使用方法上，在onCreate()和onDestroy()中需要注册和解绑，比如

```
ModuleBus.getInstance().register(
this); ModuleBus.getInstance().unregister(this);
```

最终使用类似EventBus中post方法一样，进行两个组件间的通信。这个框架的封装的post方法如下

```
public void post(Class<?> clientClass, String methodName, Object... args){if(clientClass == null || methodName == null || methodName.length() == 0) return;ArrayList<Object> clientList = getClient(clientClass,methodName);if(clientList == null) return;try{for(Objectc: clientList){try{ArrayMap<String,MethodInfo> methods = moduleEventMethods.get(c);Method method = methods.get(methodName).m;if(method == null){Log.e(TAG, "cannot find client method"+methodName + "for args\["+args.length+ "\"]"\+ Arrays.toString(args));return;}elseif(method.getParameterTypes() == null){Log.e(TAG, "cannot find client method param:"+method.getParameterTypes() + "for args\["+args.length+ "\"]"\+ Arrays.toString(args));return;}elseif(method.getParameterTypes().length != args.length){Log.e(TAG, "method "+methodName + " param number not matched:method("+method.getParameterTypes().length+ "), args("\+\ args.length+ ")");return;}method.invoke(c,args);} catch(Throwable e){Log.e(TAG, "Notify client method invoke error.",e);}}catch(Throwable e){Log.e(TAG, "Notify client error",e);}}
```

可以看到，它是通过遍历之前内部的ArrayMap，把注册在里面的方法找出，根据传入的参数进行匹配，使用反射调用。

## 接口+路由

接口+路由实现方式则相对容易理解点，我之前实践的一个项目就是通过这种方式实现的。具体地址如下：`DemoComponent` <https://github.com/cr330326/DemoComponent> 实现思路是专门抽取一个LibModule作为路由服务，每个组件声明自己提供的服务 Service API，这些 Service 都是一些接口。

组件负责将这些 Service 实现并注册到一个统一的路由 Router 中去，如果要使用某个组件的功能，只需要向Router 请求这个 Service 的实现，具体的实现细节我们全然不关心，只要能返回我们需要的结果就可以了。

比如定义两个路由地址，一个登陆组件，一个设置组件，核心代码：

```
public class RouterPath{ //注意路由的命名，路径第一个开头需要不一致，保证唯一性
 public static final String ROUTER_PATH_TO_LOGIN_SERVICE =
 "/login/service"; //Setting Service
 public static final String ROUTER_PATH_TO_SETTING_SERVICE = "/setting/service";}
```

那么就相应着就有两个接口API，如下：

```
public interface ILoginProvider extends IProvider {
 void goToLogin(Activity activity);
}
public interface ISettingProvider extends IProvider {
 void goToSetting(Activity activity);
}
```

这两个接口API对应着是向外暴露这两个组件的能力，然后每个组件对接口进行实现，如下：

```
@Route(path =
RouterPath.ROUTER_PATH_TO_LOGIN_SERVICE, name = "登陆页")
public class LoginService implements ILoginProvider {
 @Override
 public void init(Context context) {}
 @Override
 public void goToLogin(Activity activity) {
 Intent loginIntent = new Intent(activity,
 LoginActivity.class);
 activity.startActivity(loginIntent);
 }
}
```

这其中使用的到了阿里的ARouter页面跳转方式，内部本质也是接口+实现方式进行组件间通信。

调用则很简单了，如下：

```
IProvider loginService = (IProvider)
ARouter.getInstance().build(ROUTER_PATH_TO__
LOGIN_SERVICE).navigation();if(loginService != null)
{loginService.goToLogin(MainActivity. this);}
```

还有一个组件化框架，就是ModularizationArchitecture <https://github.com/SpinyTech/ModularizationArchitecture>，它本质实现方式也是接口+实现，但是封装形式稍微不一样点，它是每个功能模块中需要使用注解建立Action事件，每个Action完成一个事件动作。invoke只是方法名为反射，并未用到反射，而是使用接口方式调用，参数是通过HashMap<String, String>传递的，无法传递对象。

具体详解可以看这篇文章 Android架构思考(模块化、多进程)。

[http://blog.spinytech.com/2016/12/28/android\\_modularization/](http://blog.spinytech.com/2016/12/28/android_modularization/)

## 页面跳转

页面跳转也算是一种组件间的通信，只不过它相对粒度更细化点，之前我们描述的组件间通信粒度会更抽象点，页面跳转则是定位到某个组件的某个页面，可能是某个Activity，或者某个Fragment，要跳转到另外一个组件的Activity或Fragment，是这两者之间的通信。

甚至在一般没有进行组件化架构的工程项目中，往往也会封装页面之间的跳转代码类，往往也会有路由中心的概念。

不过一般 UI 跳转基本都会单独处理，一般通过短链的方式来跳转到具体的 Activity。

每个组件可以注册自己所能处理的短链的 Scheme 和 Host，并定义传输数据的格式，然后注册到统一的 UIRouter 中，UIRouter 通过 Scheme 和 Host 的匹配关系负责分发路由。但目前比较主流的做法是通过在每个 Activity 上添加注解，然后通过 APT 形成具体的逻辑代码。

下面简单介绍目前比较主流的两个框架核心实现思路：

## ARouter

ARouter 核心实现思路是，我们在代码里加入的@Route注解，会在编译时期通过apt生成一些存储path和activityClass映射关系的类文件。

然后app进程启动的时候会拿到这些类文件，把保存这些映射关系的数据读到内存里(保存在map里)。

然后在进行路由跳转的时候，通过build()方法传入要到达页面的路由地址，ARouter会通过它自己存储的路由表找到路由地址对应的Activity.class(activity.class = map.get(path))，然后new Intent()，当调用ARouter的withString()方法它的内部会调用intent.putExtra(String name, String value)，调用navigation()方法，它的内部会调用startActivity(intent)进行跳转，这样便可以实现两个相互没有依赖的module顺利的启动对方的Activity了。

## ActivityRouter

ActivityRouter 核心实现思路是，它是通过路由 + 静态方法来实现，在静态方法上加注解来暴露服务，但不支持返回值，且参数固定位(context, bundle)，基于apt技术，通过注解方式来实现URL打开Activity功能，并支持在WebView和外部浏览器使用，支持多级Activity跳转，支持Bundle、Uri参数注入并转换参数类型。

它实现相对简单点，也是比较早期比较流行的做法，不过学习它也是很有参考意义的。

## 5.小结

总的来说，组件间的通信机制在组件化编程和组件化架构中是很重要的一个环节，可能在每个组件独自开发阶段，不需要与其他组件进行通信，只需要在内部通信即可。

当处于组件集成阶段，那就需要大量组件进行互相通信，体现在每个业务互相协作，如果组件间设计的不好，打开一个页面或调用一个方法，想当耗时或响应慢，那么体现的则是这个APP使用比较卡顿，仅仅打开一个页面就是需要好几秒才能打开，则严重影响使用者的体验了，甚至一些大型APP，可能组件分化更小，种类更多，那么组件间的通信则至关重要了。所以，要打造一个良好的组件化框架，如何设计一个更适合自己的业务类型的通信机制，就需要多多进行思考了。

## 20.4.8 ARouter系列三：依赖注入暴露服务

如果对于ARouter不是很了解的，建议先行阅读：

[ARouter系列一：Activity跳转原理详解](#)

[ARouter系列二：@Autowired属性注入](#)

想要提供服务暴露给其他人使用，ARouter提供了一个接口：`IProvider`

```
// Provider interface, base of other interface.public
interface IProvider { /** * Do your init work in
this method, it well be call when processor has been
load. * * @param context ctx */ void
init(Context context);}
```

`IProvider` 只有一个方法 `init`，参数为 `Context`。刚看到这接口的时候不是很明白，等把本篇文章看完之后就会明白，参数 `context` 不可或缺。注意 `IProvider` 的注释：`IProvider` 是其它 Interface 的基础。可见 ARouter 推荐的用法是一个 Interface 继承 `IProvider`。

这里大家可能会有疑问，为什么需要 `Context` 参数，为什么推荐是一个接口继承 `IProvider`，而不是实体类。带着这些疑问，我们继续探寻

### 一、使用

新建一个项目，有两个 module：`app` 和 `location`。`location` 中封装了地图相关服务，例如：开始定位，获取位置等等。`app` 依赖 `location`

#### 1.1 location模块

`LocationService` 定义此模块提供的基础服务：开始定位和获取地理位置。并且实现了上述的 `IProvider`

```
package com.daddyno1.location;public interface
LocationService extends IProvider { //开始定位 void
startLocate(); //获取经纬度信息 Point getLocation();}
```

`BaiduLocationService` 是提供服务的最终实现类。此处模拟使用了百度来实现具体功能。并且使用了 `@Route(path = "/Loc/LocationService")` 来标注真实 `Location` 服务实现类。

```
@Route(path = "/Loc/LocationService")public class
BaiduLocationService implements LocationService {
private static final String TAG = "BaiduLocationService";
 @Override public void startLocate() {
Log.i(TAG, "BaiduLocationService - startLocate");
 }
@Override public Point getLocation() {
 return new Point(11,12);
}
@Override public void init(Context context) {
 Log.i(TAG,
"BaiduLocationService - init");
}}
```

编译一下，我们看一下 apt 帮我们生成了哪些 **辅助类**：

`ARouter$$Root$$location`、`ARouter$$Group$$Loc` 和  
[ARouter\\$\\$Providers\\$\\$location](#)，[ARouter系列一：Activity跳转原理详解](#) 的不难理解，`ARouter$$Root$$location` 存储了 **location** module 下所有路由组信息；`ARouter$$Group$$Loc` 存储了 **location** module 下路由组名为 `Loc` 的路由表信息。

```
package com.alibaba.android.arouter.routes;.../** * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/
public class ARouter$$Root$$location implements
IRouteRoot {
 @Override public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
 routes.put("Loc", ARouter$$Group$$Loc.class);
 }
}
```

```
package com.alibaba.android.arouter.routes;.../** * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/
public class ARouter$$Group$$Loc implements IRouteGroup {
 @Override public void loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/Loc/LocationService",
RouteMeta.build(RouteType.PROVIDER,
BaiduLocationService.class, "/loc/locationservice",
"Loc", null, -1, -2147483648));
 }
}
```

`ARouter$$Providers$$location` 这个类长什么样？我们来看一下：

```
package com.alibaba.android.arouter.routes;.../* * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/public class ARouter$$Providers$$location implements
IProviderGroup { @Override public void
loadInto(Map<String, RouteMeta> providers) {
providers.put("com.daddyno1.location.LocationService",
RouteMeta.build(RouteType.PROVIDER,
BaiduLocationService.class, "/Loc/LocationService",
"Loc", null, -1, -2147483648)); }}
```

`ARouter$$Providers$$location` 这个类有什么用呢，我们先往后看，一会再说。

## 1.2 app模块

这里新建一个 Activity 使用我们创建好的 **Location** 服务

```
public class SecondActivity extends AppCompatActivity {
 @Override protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main3); // 使用一:
 by name LocationService locationService =
 (LocationService)
 ARouter.getInstance().build("/Loc/LocationService").navi
 gation(); locationService.startLocate(); //使
 用二: by type LocationService locationService1 =
 ARouter.getInstance().navigation(LocationService.class);
 locationService1.startLocate(); }}
```

我们运行一下，看一下结果：

```
I: BaiduLocationService - initI: BaiduLocationService -
startLocateI: BaiduLocationService - startLocate
```

注意： init 执行了一次， startLocate 调用了两次。

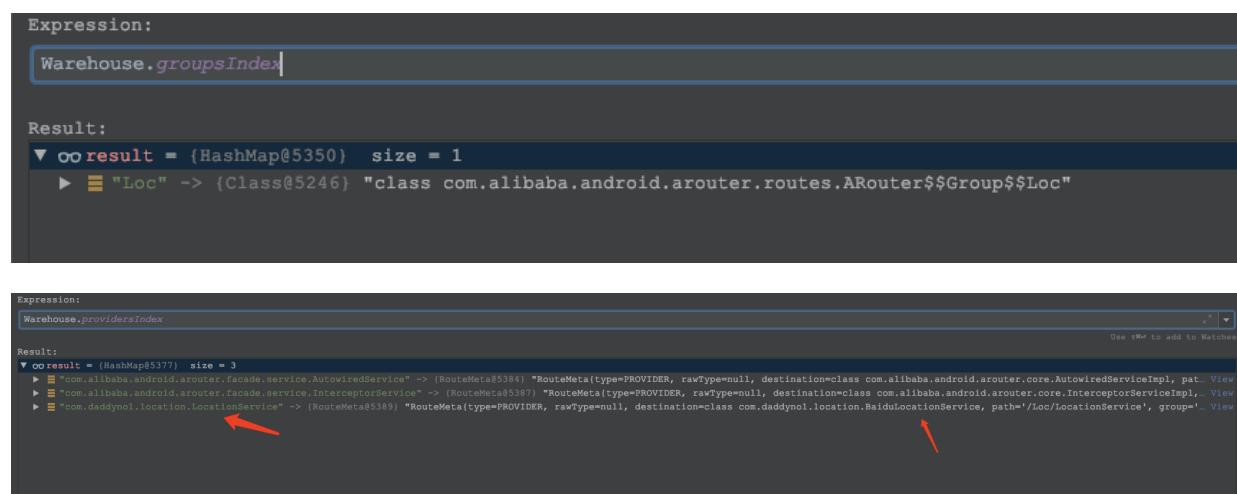
## 二、源码分析

### 2.1 init

之前在分析Activity跳转的时候分析了ARouter的初始化，这里不再细讲。

```
public synchronized static void init(Context context,
ThreadPoolExecutor tpe) throws HandlerException { ... for
(String className : routerMap) { if
(className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME
+ SEPARATOR + SUFFIX_ROOT)) {
//com.alibaba.android.arouter.routes.ARouter$$Root //
This one of root elements, load root. ((IRouteRoot)
Class.forName(className).getConstructor().newInstance())
.loadInto(warehouse.groupsIndex); } else if
(className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME
+ SEPARATOR + SUFFIX_INTERCEPTORS)) {
//com.alibaba.android.arouter.routes.ARouter$$Interceptors
// Load interceptorMeta ((IInterceptorGroup)
Class.forName(className).getConstructor().newInstance()
.loadInto(warehouse.interceptorsIndex); } else if
(className.startsWith(ROUTE_ROOT_PAKCAGE + DOT + SDK_NAME
+ SEPARATOR + SUFFIX_PROVIDERS)) {
//com.alibaba.android.arouter.routes.ARouter$$Providers
// Load providerIndex ((IProviderGroup)
Class.forName(className).getConstructor().newInstance()
.loadInto(warehouse.providersIndex); } } ... }
```

唯一需要提一下的是，经过路由器初始化之后，Warehouse的数据如下：



`ARouter$$Providers$$location` 这个是之前提到的apt生成的辅助类，它的作用就是帮助路由在初始化的时候，把相关数据注入到 `Warehouse.providersIndex`，这个有什么用呢，其实对于 `by name` 的使用方式来说，`warehouse.providersIndex` 和 `ARouter$$Providers$$location` 根本没用上，只是 `by type` 时候的辅助类。

## 2.2 by name

```
LocationService locationService = (LocationService)
ARouter.getInstance().build("/Loc/LocationService").navigation();
```

## 2.3 by type

```
LocationService locationService1 =
ARouter.getInstance().navigation(LocationService.class);
```

## 2.4 小结

具体的实现逻辑和 Activity 路由过程很相似，自己去看代码吧。唯一的区别是，如果 `RouteType` 是 `PROVIDER` 的话，直接根据反射创建一个路由 `Destination` 类的实例，存到 `Postcard` 里，最后返回给用户。

本例中 `ARouter` 会使用反射创建一个 `BaiduLocationService` 的实体对象，回调 `init` 方法，返回给使用者，之后把 实体对象缓存到 `Warehouse.providers` (`Map<Class, IProvider>`)，下次直接在使用的话，直接从缓存获取，不会再反射创建了。这也是本例为什么 `init` 只执行了一次。

## 三、总结

### 1、为什么 `IProvider` 要提供一个 `Context`？

我们知道 `ARouter` 是帮助我们实现项目组件化的工具，很多情况下一个 `Moudle` 是拿不到应用程序 `Context` 的，本例中的场景就是。但很多服务的实现又会依赖于 `Context`，所以有必要把 `Context` 传递给各个 `Moudle`。

## 2、为什么推荐 Interface 集成 IPriovider?

ARouter说 通过依赖注入暴露服务。本例中抽象了一个 LocationService 接口，此接口抽象了所有用到的服务，用户通过ARouter可以拿到 LocationService 的具体实现类，然后强转成 LocationService 使用，可以不用关心具体的实现类是谁。这就是通过依赖注入的方式 实现 控制翻转，从而实现使用者 和 具体实现解耦的目的。假如某一天又想时候用 高德SDK 实现具体的 Location 服务，使用者的代码不用修改，实现完全的解耦。使用 高德 实现 Location 服务：

```
package com.daddyno1.location;...@Route(path =
"/Loc/LocationService")public class GaodeLocationService
implements LocationService { private static final
String TAG = "GaodeLocationService"; @Override
public void startLocate() { Log.i(TAG, "
GaodeLocationService - startLocate"); } @Override
public Point getLocation() { return new
Point(1,2); } @Override public void init(Context
context) { Log.i(TAG, " GaodeLocationService -
init"); }}
```

### 20.4.9 ARouter: Activity跳转原理详解

本篇文章默认读者已经会使用 ARouter，我们通过使用代码及源码详  
解其原理。

ARouter是一个路由框架。使用AndroidStudio 开发环境，我们知道在 Coding阶段，没有依赖关系的Module，它们的类是无法被对方直接引用。这是AS 的项目组织方式。但是不管所属哪一个Moudle的代码，最终都会打包进apk，在软件运行时，我们可以拿得所有的类。

**现在有一个问题：两个没有依赖关系的Moudle，它们之间的页面该如何跳转呢？**

既然Coding阶段无法拿到对象，那我们就运行时获取到进行页面跳转所需的信息。

假如有 module1 和 module2，之间没有依赖关系。AActivity (包名：

com.test) 属于 moudle1; BActivity (包名: con.test) 属于 module2。从 **A** 页面跳转到 **B** 页面, 我们可以使用如下方式:

```
startActivity(new Intent(this,
Class.forName("com.test.BAcctivity"))或者Intent i = new
Intent();i.setComponent(new ComponentName("com.test",
"com.test.BActivity"));startActivity(i);
```

ARouter的解决方案是, 全局维护一个Map集合, key是一段用户定义的字符串 (本例中的a, b) , value是对应的类对象。类似这种 (概念上) :

```
Map<String, Class> map = new HashMap<String, Class>
();map.put("a", AActivity.class);map.put("b",
BActivity.class);
```

我们在开发阶段只用使用a, b就可以代表 AActivity, BActivity, 然后在运行阶段通过映射关系进行转换后创建 Intent 对象进行跳转。

(Intent(Context packageContext, Class<?> cls))

使用的时候类似 (概念上) :

```
ARouter.getInstance().build("b").navigation();
```

接下来探寻一下, ARouter的具体实现。看ARouter的源码能学到: 框架的设计、APT、Gradle-Plugin。

## 一、项目目录结构概述

|                       |                                                            |               |
|-----------------------|------------------------------------------------------------|---------------|
| .github               | Official Gradle Wrapper Validation Action                  | 4 months ago  |
| app                   | Fix wrong spelling                                         | 6 months ago  |
| arouter-annotation    | Unified compiler and api sdk version.                      | 16 months ago |
| arouter-api           | Correct function name.                                     | 3 months ago  |
| arouter-compiler      | Support for parsing url parameters that jump to Fragment.  | 13 days ago   |
| arouter-gradle-plugin | fix repeated inject init code when Multi-channel packaging | 17 months ago |
| arouter-idea-plugin   | Set plugin init version                                    | 2 years ago   |
| demo                  | Update demo app                                            | 2 years ago   |
| gradle/wrapper        | Update gradle plugin and dependencies.                     | 2 years ago   |
| module-java           | Update gradle plugin and dependencies.                     | 2 years ago   |
| module-kotlin         | Update gradle plugin and dependencies.                     | 2 years ago   |
| .gitignore            | Fix wrong spelling                                         | 6 months ago  |
| LICENSE               | Initial commit                                             | 4 years ago   |

## 项目结构

`arouter-annotation`: 定义注解类

`arouter-api` : 核心库，封装了开发人员使用的API

`arouter-compiler`: 定义编译时注解处理器，通过扫描项目，生成辅助Java类

`arouter-gradle-plugin`: gradle插件（可选），用于在编译时通过Transform处理字节码

## 二、注解

```
/** * Mark a page can be route by router.
*/@Target({ElementType.TYPE})@Retention(RetentionPolicy.CLASS)public @interface Route { /** * Path of route */ String path(); /** * Used to merger routes, the group name MUST BE USE THE COMMON WORDS !!! */ String group() default ""; ...}
```

`Route` 注解类，是一个标记，被标记的实体能够被路由进行分发，例如：`Activity`。`path`就是将来路由表中的 "key"，`group` 代表由`Route`标记的实体 所属的路由组。

路由组的概念：把所有的路由表信息，分成N组，这样在加载路由表的时候，可以分组加载，提高效率。即按需加载，有延迟加载的特性，同时又可以减少路由表加载时间。

## 三、注解处理器

项目编译时通过扫描注解，根据这些有用的信息，动态生成辅助的Java类文件`.java`。`RouteProcessor.java` 注解处理器就是为了处理那些被`Route`标注的类。因为需要动态生成`.java`文件，项目中使用了`javapoet` 作为辅助工具，以提高生产效率。

注解处理器生成的“**辅助的文件**”，也可以我们自己手动编写。但通常情况是，这些`java`文件中的逻辑单一简单，且有一定的共同特性（这需要大家在开发过程中去观察），所以通过注解信息去生成相关的`.java` 可以大大提高生产力。`ARouter`、`ButterKnife` 中大量使用了此技术。

这里不打算对 `RouteProcessor.java` 的细节进行介绍，有兴趣的可以去看代码。过度讲解细节，会让主线模糊不清。（当然这些细节也很重要，可以考虑再开一节来深入细节）

我们结合具体的代码，看一下帮助我们生成了哪些“辅助类”

新建一个空的Android项目，只有一个默认的module: `app`，创建两个页面：`FirstActivity` 和 `SecondActivity`（这里省略ARouter 依赖的引用以及库的初始化）

```
package com.daddyno1.projectmoduledemo;...@Route(path =
"/group1/first") public class FirstActivity extends
AppCompatActivity { @Override protected void
onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main2); } } package
com.daddyno1.projectmoduledemo;...@Route(path =
"/group2/second")public class SecondActivity extends
AppCompatActivity { @Override protected void
onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main3); }}
```

这两个 Activity 的path设定，特意分属两个路由组，group1 和 group2（会截取path 前两个/ 的字符串作为组）

编译一下，我们可以看看生成了哪些辅助文件。（辅助文件的位置在 `app/build/generated/ap_generated_sources` 下），生成的路由辅助文件包名：`com.alibaba.android.arouter.routes`



APT 生成的辅助类

`ARouter$$Root$$xxx`: xxx 为本module的名称。本例中module名称是 `app`，此类记录了 `app` 中所有路由组信息。

`ARouter$$Group$$xxx`: xxx为本module中路由组的名称。本例中共有2个分组: group1 和 group2。此类记录了某个路由组中所有的路由表信息。

```
package com.alibaba.android.arouter.routes;.../** * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/public class ARouter$$Root$$app implements IRouteRoot {
 @Override public void loadInto(Map<String, Class<?
 extends IRouteGroup>> routes) { routes.put("group1",
 ARouter$$Group$$group1.class); routes.put("group2",
 ARouter$$Group$$group2.class); } }
```

```
package com.alibaba.android.arouter.routes;.../** * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/public class ARouter$$Group$$group1 implements
IRouteGroup { @Override public void
loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/group1/first",
 RouteMeta.build(RouteType.ACTIVITY, FirstActivity.class,
 "/group1/first", "group1", new java.util.HashMap<String,
 Integer>(){put("paramInt", 3); }, -1, -2147483648));
}
```

```
package com.alibaba.android.arouter.routes;.../** * DO
NOT EDIT THIS FILE!!! IT WAS GENERATED BY AROUTER.
*/public class ARouter$$Group$$group2 implements
IRouteGroup { @Override public void
loadInto(Map<String, RouteMeta> atlas) {
 atlas.put("/group2/second",
 RouteMeta.build(RouteType.ACTIVITY, SecondActivity.class,
 "/group2/second", "group2", null, -1, -2147483648)); } }
```

根据代码我们可以看到 `ARouter$$Group$$xxx` 中路由表信息集合的 `key` 就是之前用 `@Route` 注解的 `path` 参数, 如 `/group1/first` 和 `/group/second`。`value` 就是一个 `RouteMeta` 对象, 即路由的元数据, 此对象记录了本次路由映射需要的最基本必要的信息。

```
public class RouteMeta { ... private RouteType
type; // Type of route private Class<?>
destination; // Destination private String path;
// Path of route private String group;
// Group of route ...
```

至此，至此我们了解到这些辅助文件的生成规则。

## 四、核心库的设计

接下来就是核心库的部分，之前的动态生成了那么多“**辅助类**”，现在看一下该怎么使用。

### 4.1 初始化

`ARouter.init(this);` 在Application中调用 **init** 方法进行初始化路由。

```
public final class ARouter { /** * 初始化。在使用路
由之前必须先调用 */ public static void
init(Application application) { if (!hasInit) {
... hasInit =
_ARouter.init(application); ... } }}
```

```
final class _ARouter { protected static synchronized
boolean init(Application application) { ...
LogisticsCenter.init(mContext, executor); ...
}}
```

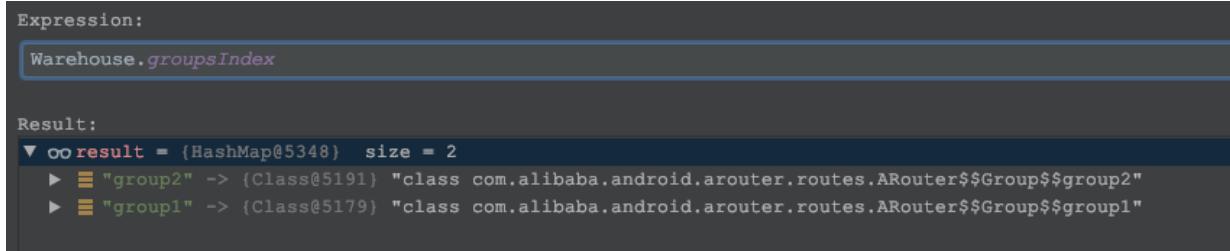
以上就是最基本的 **门面设计模式** 的应用。核心的初始化工作在 `LogisticsCenter` 中

```
public class LogisticsCenter { public synchronized
static void init(Context context, ThreadPoolExecutor tpe)
throws HandlerException { Set<String> routerMap;
 // 遍历dex文件，找到
com.alibaba.android.arouter.routes 下被 arouter-compiler
创建的辅助类 routerMap =
Classutils.getFileNameByPackageName(mContext,
ROUTE_ROOT_PAKCAGE); ... for (String
className : routerMap) {
//com.alibaba.android.arouter.routes.ARouter$$Root
 if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT
+ SDK_NAME + SEPARATOR + SUFFIX_ROOT)) {
 // This one of root elements, load root.
 ((IRouteRoot)
(Class.forName(className).getConstructor().newInstance())
.loadInto(warehouse.groupsIndex)); } else
if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT +
SDK_NAME + SEPARATOR + SUFFIX_INTERCEPTORS)) {
 // Load interceptorMeta
 ((IInterceptorGroup)
(Class.forName(className).getConstructor().newInstance())
.loadInto(warehouse.interceptorsIndex)); }
else if (className.startsWith(ROUTE_ROOT_PAKCAGE + DOT +
SDK_NAME + SEPARATOR + SUFFIX_PROVIDERS)) {
 // Load providerIndex
 ((IProviderGroup)
(Class.forName(className).getConstructor().newInstance())
.loadInto(warehouse.providersIndex)); }
 } }}
```

以上代码是简化版本，核心逻辑是扫描dex文件中 APT 生成的辅助类，**因为本篇讲的是 Activity 跳转，所以会忽略其他部分的内容**。类的全类名以 `com.alibaba.android.arouter.routes.ARouter$$Root` 开头的，将会被通过反射构建对象实体，然后调用其 `LoadInto` 方法把路由分组信息注入到 `warehouse.groupsIndex`。`Warehouse` 是缓存了所有路由信息的工具类。

```
class Warehouse { //路由组信息缓存 static Map<String, Class<? extends IRouteGroup>> groupsIndex = new HashMap<>(); //路由表信息缓存 static Map<String, RouteMeta> routes = new HashMap<>(); ...}
```

经过 `init` 初始化以后，`Warehouse.groupsIndex` 缓存了项目中所有路由组信息：



- 1、这里让大家思考一个问题，既然路由初始化时，会扫描 dex 文件，获取指定包下的类信息，假如APK被加固了，正确的dex文件被隐藏，就无法找到相关的辅助类，该如何处理？
- 2、`LogisticsCenter#init` 初始化中，也使用到了缓存，从dex文件扫描获取到的类信息会缓存在SP里，下次就直接从SP里获取了，不必扫描 dex文件。而且这里还有版本控制，如果是新版本的话，就会重新扫描dex文件，之后更新SP里的信息。

## 4.2 使用路由进行页面跳转

```
ARouter.getInstance().build("/group1/first").navigation();
```

这行代码就是 跳转到 `/group1/first` 在路由表中所标识的真实地址：`FirstActivity`（参考之前APT 生成的辅助类）。其实很好想明白怎么实现，执行 `navigation` 方法时，根据路由表的信息，获取跳转的目标地址：`FirstActivity.class`，然后构造 `Intent`，执行`startActivity`方法，即可完成跳转。

在分析代码之前，我们先来了解一个类 `Postcard`，中文名字叫明信片，继承自 `RouteMeta`，除了包含最基本的路由信息外，还有一些其他信息，比如参数传递的 `Bundle` 对象、动画、`IntentFlag`等等，**这个类是进行路由分发的基本数据组成单元。**（注意区分 `Postcard` 和 `RouteMeta`）

```
/** * A container that contains the roadmap. */public
final class Postcard extends RouteMeta { // Base
private Uri uri; private Object tag; // A
tag prepare for some thing wrong. private Bundle
mBundle; // Data to transform private int
flags = -1; // Flags of route private int
timeout = 300; // Navigation timeout,
TimeUnit.Second private IProvider provider; // It
will be set value, if this postcard was provider.
private boolean greenChannel; private
SerializationService serializationService; //
Animation private Bundle optionsCompat; // The
transition animation of activity private int
enterAnim; private int exitAnim;}
```

ARouter.getInstance().build("/group1/first") 此时就会构造一个 Postcard 对象，此时对象属性只有 path 和 group 是确定的，之后初始化 Postcard 对象 (LogisticsCenter#completion)，然后使用 Postcard 进行导航，如果 RouteType 是 ACTIVITY，则会构建 Intent，调用 ActivityCompat#startActivity 启动页面：

想  
識  
思  
學

```
final class _ARouter { ... protected Object
navigation(final Context context, final Postcard
postcard, final int requestCode, final NavigationCallback
callback) { try { // 完善 Postcard 信息
LogisticsCenter.completion(postcard); }
catch (NoRouteFoundException ex) { ... //未找
到路由信息的处理逻辑 return null; }
if (null != callback) {
callback.onFound(postcard); } if
(!postcard.isGreenChannel()) { // It must be run in
async thread, maybe interceptor cost too mush time made
ANR. ... //拦截器相关处理 } else {
//导航 return _navigation(context, postcard,
requestCode, callback); } return null; }
// 进行导航 private Object _navigation(final
Context context, final Postcard postcard, final int
requestCode, final NavigationCallback callback) {
final Context currentContext = null == context ? mContext
: context; switch (postcard.getType()) {
case ACTIVITY: //如果是 Activity // 构造
Intent final Intent intent = new
Intent(currentContext, postcard.getDestination());
 intent.putExtras(postcard.getExtras()); //设置
bundle // 设置 flags int
flags = postcard.getFlags(); if (-1 !=
flags) { intent.setFlags(flags);
} else if (!(currentContext instanceof
Activity)) { // Non activity, need less one flag.
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
} // Navigation in main looper.
new Handler(Looper.getMainLooper()).post(new
Runnable() { @Override
public void run() { if
(requestCode > 0) { // Need start for result
ActivityCompat.startActivityForResult((Activity)
currentContext, intent, requestCode,
```

```
postcard.getOptionsBundle()); }
else {
 ActivityCompat.startActivity(currentContext, intent,
 postcard.getOptionsBundle()); }
 //设置出入动画
if ((0 != postcard.getEnterAnim() || 0 !=
 postcard.getExitAnim()) && currentContext instanceof
 Activity) { // old version.
 ((Activity)
 currentContext).overridePendingTransition(postcard.getEnt
 erAnim(), postcard.getExitAnim());
}
 //导航回调
if (null != callback) { // Navigation over.
 callback.onArrival(postcard);
}
}
break;
... } }
return null; }}
```

从上边的分析可知，每次进行 navigation 的时候都要创建一个 Postcard，然后根据路由表的信息，对 Postcard 对象进行初始化，之后进行真正的 `navigation()`。我们接下来看一下，如何对 Postcard 对象进行初始化：`LogisticsCenter#completion`

```

public synchronized static void completion(Postcard
postcard) { ... //根据path获取路由信息
RouteMeta routeMeta =
Warehouse.routes.get(postcard.getPath()); //如果路由
信息为空，则可能此分组的路由表还未加载过；当然也可能是压根就找不到
路由信息。 if (null == routeMeta) {
// 获取分组信息 class<? extends IRouteGroup>
groupMeta =
Warehouse.groupsIndex.get(postcard.getGroup());
//如果分组信息不存在，则抛出异常。有可能是开始就没有分组信息，也有可能是 路由分组信息加载过了之后会被清理。 if (null ==
groupMeta) {
throw new
NoRouteNotFoundException(TAG + "There is no route match the
path [" + postcard.getPath() + "], in group [" +
postcard.getGroup() + "]"); } else {
try { //加载此分组的路由信息到路由表，然后把此路由分组删除。
IRouteGroup
iGroupInstance =
groupMeta.getConstructor().newInstance();
iGroupInstance.loadInto(Warehouse.routes);
Warehouse.groupsIndex.remove(postcard.getGroup());
//如果不删除，假设navigation一个分组中没有的路由表信息，就会无限循环找下去了。其实只要路由分组表被加载过一次，路由表（Warehouse.routes）就会包含此分组的路由表。
} catch (Exception e) {
throw new HandlerException(TAG + "Fatal exception
when loading group meta. [" + e.getMessage() + "]");
}
completion(postcard); // 路由分组的路由表被加载以后，调用初始化方法
}
else { //找到路由信息，初始化Postcard
postcard.setDestination(routeMeta.getDestination());
postcard.setType(routeMeta.getType());
postcard.setPriority(routeMeta.getPriority());
postcard.setExtra(routeMeta.getExtra()); Uri
rawUri = postcard.getUri(); ...
}
}

```

至此，一次完整的页面跳转的处理过程分析完成。

## 五、Gradle-Plugin (可选)

之前有一个问题是，加固以后APK找不到，路由信息的这些类了，怎么办？通过以上流程分析，我们可知 `ARouter.init` 初始化路由的时候，会去扫描 dex 文件，找到路由信息的类，然后装载 路由分组类 到内存。`gradle-plugin` 的作用 在编译期间动态修改字节码文件，动态插入到 `LogisticsCenter` 类中，改变init的逻辑。当然这是一个可选的模块。

使用`gradle-plugin`之前，`init` 的代码逻辑一定会执行 dex 扫描的逻辑，因为 `loadRouterMap` 方法为空，`registerByPlugin` 永远为`false`。代码详情：

```
/** * arouter-auto-register plugin will generate
code inside this method * call this method to
register all Routers, Interceptors and Providers *
@author billy.qi Contact me. *
@since 2017-12-06 */ private static void
loadRouterMap() { registerByPlugin = false;
//auto generate register code by gradle plugin: arouter-
auto-register // looks like below: //
registerRouteRoot(new ARouter..Root..modulejava());
// registerRouteRoot(new ARouter..Root..modulekotlin());
} public synchronized static void init(Context
context, ThreadPoolExecutor tpe) throws HandlerException
{
 try {
 long startInit =
System.currentTimeMillis(); //billy.qi
modified at 2017-12-06 //load by plugin first
 loadRouterMap(); if
(registerByPlugin) { logger.info(TAG,
"Load router map by arouter-auto-register plugin.");
} else { //扫描 dex, 初始化路由分组信息
 }
}
```

使用了 `gradle-plugin` 之后，我们把生成的apk反编译后，看一下真实的代码是什么样的：

```
com.alibaba.android.arouter.core
 ↳ ARouter.class
 ↳ InterceptorServiceImpl.class
 ↳ LogisticsCenter.class
 ↳ Warehouse.class
 ↳ exception
 ↳ facade
 ↳ annotation
 ↳ callback
 ↳ enums
 ↳ model
 ↳ service
 ↳ template
 ↳ IInterceptor.class
 ↳ IInterceptorGroup.class
 ↳ ILogger.class
 ↳ IPolicy.class
 ↳ IProvider.class
 ↳ IProviderGroup.class
 ↳ IRouteGroup.class
 ↳ IRouteRoot.class
 ↳ IRouteRoot.class
 ↳ IRouteRoot.class

private static void loadRouterMap()
{
 registerByPlugin = false;
 register("com.alibaba.android.arouter.routes.ARouter$$Root$$arouterapi");
 register("com.alibaba.android.arouter.routes.ARouter$$Root$$app"); ←
 register("com.alibaba.android.arouter.routes.ARouter$$Providers$$arouterapi");
 register("com.alibaba.android.arouter.routes.ARouter$$Providers$$app");
}

private static void markRegisteredByPlugin()
{
 if (!registerByPlugin) {
 registerByPlugin = true;
 }
}
```

会在执行 `loadRouterMap` 方法的时候，去装载所有路由分组信息，之后 `registerByPlugin` 被设置成`true`，接下来逻辑判断的时候就不会继续使用 dex 扫描的方式。

至此，ARouter 的全貌已浮出水面，对它也有了一个更清晰的认识。

## 20.5 算法设计

### 20.5.1 时间复杂度

#### 一、算法的时间与空间复杂度（一看就懂）

算法（Algorithm）是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别。

那么我们应该如何去衡量不同算法之间的优劣呢？

主要还是从算法所占用的「时间」和「空间」两个维度去考量。

- **时间维度：**是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
- **空间维度：**是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」来描述。

因此，评价一个算法的效率主要是看它的时间复杂度和空间复杂度情况。然而，有的时候时间和空间却又是「鱼和熊掌」，不可兼得的，那么我们就需要从中去取一个平衡点。

下面我来分别介绍一下「时间复杂度」和「空间复杂度」的计算方式。

## 一、时间复杂度

我们想要知道一个算法的「时间复杂度」，很多人首先想到的方法就是把这个算法程序运行一遍，那么它所消耗的时间就自然而然知道了。

这种方式可以吗？当然可以，不过它也有很多弊端。

这种方式非常容易受运行环境的影响，在性能高的机器上跑出来的结果与在性能低的机器上跑的结果相差会很大。而且对测试时使用的数据规模也有很大关系。再者，并我们在写算法的时候，还没有办法完整的去运行呢。

因此，另一种更为通用的方法就出来了：「**大O符号表示法**」，即  $T(n) = O(f(n))$

我们先来看个例子：

```
for(i=1; i<=n; ++i){ j = i; j++;}
```

通过「大O符号表示法」，这段代码的时间复杂度为： $O(n)$ ，为什么呢？

在大O符号表示法中，时间复杂度的公式是： $T(n) = O(f(n))$ ，其中  $f(n)$  表示每行代码执行次数之和，而  $O$  表示正比例关系，这个公式的全称是：**算法的渐进时间复杂度**。

我们继续看上面的例子，假设每行代码的执行时间都是一样的，我们用 1 颗粒时间 来表示，那么这个例子的第一行耗时是 1 个颗粒时间，第三行的执行时间是  $n$  个颗粒时间，第四行的执行时间也是  $n$  个颗粒时间（第二行和第五行是符号，暂时忽略），那么总时间就是 1 颗粒时间 +  $n$  颗粒时间 +  $n$  颗粒时间，即  $(1+2n)$  个颗粒时间，即： $T(n) = (1+2n)*\text{颗粒时间}$ ，从这个结果可以看出，这个算法的耗时是随着  $n$  的变化而变化，因此，我们可以简化的将这个算法的时间复杂度表示为： $T(n) = O(n)$

为什么可以这么去简化呢，因为大O符号表示法并不是用来真实代表算法的执行时间的，它是用来表示代码执行时间的增长变化趋势的。

所以上面的例子中，如果  $n$  无限大的时候， $T(n) = \text{time}(1+2n)$  中的常量 1 就没有意义了，倍数 2 也意义不大。因此直接简化为  $T(n) = O(n)$  就可以了。

常见的时间复杂度量级有：

- 常数阶O(1)
- 对数阶O(logN)
- 线性阶O(n)
- 线性对数阶O(nlogN)
- 平方阶O(n<sup>2</sup>)
- 立方阶O(n<sup>3</sup>)
- K次方阶O(n<sup>k</sup>)
- 指数阶(2<sup>n</sup>)

上面从上至下依次的时间复杂度越来越大，执行的效率越来越低。

下面选取一些较为常用的来讲解一下（没有严格按照顺序）：

### 1. 常数阶O(1)

无论代码执行了多少行，只要是没有任何循环等复杂结构，那这个代码的时间复杂度就都是O(1)，如：

```
int i = 1; int j = 2; ++i; j++; int m = i + j;
```

上述代码在执行的时候，它消耗的时候并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用O(1)来表示它的时间复杂度。

### 1. 线性阶O(n)

这个在最开始的代码示例中就讲解过了，如：

```
for(i=1; i<=n; ++i){ j = i; j++;}
```

这段代码，for循环里面的代码会执行n遍，因此它消耗的时间是随着n的变化而变化的，因此这类代码都可以用O(n)来表示它的时间复杂度。

### 1. 对数阶O(logN)

还是先来看代码：

```
int i = 1; while(i<n){ i = i * 2;}
```

从上面代码可以看到，在while循环里面，每次都将 i 乘以 2，乘完之后，i 距离 n 就越来越近了。我们试着求解一下，假设循环x次之后，i 就大于 2 了，此时这个循环就退出了，也就是说 2 的 x 次方等于 n，那么  $x = \log_2 n$

也就是说当循环  $\log_2 n$  次以后，这个代码就结束了。因此这个代码的时间复杂度为：**O(logn)**

## 1. 线性对数阶O(nlogN)

线性对数阶O(nlogN) 其实非常容易理解，将时间复杂度为O(logn)的代码循环N遍的话，那么它的时间复杂度就是  $n * O(\log n)$ ，也就是了 O(nlogN)。

就拿上面的代码加一点修改来举例：

```
for(m=1; m<n; m++){ i = 1; while(i<n) { i = i * 2; }}
```

## 1. 平方阶O(n<sup>2</sup>)

平方阶O(n<sup>2</sup>) 就更容易理解了，如果把 O(n) 的代码再嵌套循环一遍，它的时间复杂度就是 O(n<sup>2</sup>) 了。

举例：

```
for(x=1; i<=n; x++){ for(i=1; i<=n; i++) { j = i; j++; }}
```

这段代码其实就是嵌套了2层n循环，它的时间复杂度就是 O(n\*n)，即 O(n<sup>2</sup>)

如果将其中一层循环的n改成m，即：

```
for(x=1; i<=m; x++){ for(i=1; i<=n; i++) { j = i; j++; }}
```

那它的时间复杂度就变成了 O(m\*n)

## 1. 立方阶O(n<sup>3</sup>)、K次方阶O(n<sup>k</sup>)

参考上面的O(n<sup>2</sup>) 去理解就好了，O(n<sup>3</sup>)相当于三层n循环，其它的类似。

除此之外，其实还有 平均时间复杂度、均摊时间复杂度、最坏时间复杂度、最好时间复杂度 的分析方法，有点复杂，这里就不展开了。

## 二、空间复杂度

既然时间复杂度不是用来计算程序具体耗时的，那么我也应该明白，空间复杂度也不是用来计算程序实际占用的空间的。

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的一个量度，同样反映的是一个趋势，我们用  $S(n)$  来定义。

空间复杂度比较常用的有： $O(1)$ 、 $O(n)$ 、 $O(n^2)$ ，我们下面来看看：

### 1. 空间复杂度 $O(1)$

如果算法执行所需要的临时空间不随着某个变量n的大小而变化，即此算法空间复杂度为一个常量，可表示为  $O(1)$

举例：

```
int i = 1; int j = 2; ++i; j++; int m = i + j;
```

代码中的  $i$ 、 $j$ 、 $m$  所分配的空间都不随着处理数据量变化，因此它的空间复杂度  $S(n) = O(1)$

### 1. 空间复杂度 $O(n)$

我们先看一个代码：

```
int[] m = new int[n] for(i=1; i<=n; ++i){ j = i; j++;}
```

这段代码中，第一行new了一个数组出来，这个数据占用的大小为n，这段代码的2-6行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即  $S(n) = O(n)$

以上，就是对算法的时间复杂度与空间复杂度基础的分析，欢迎大家一起交流。

## 20.5.2 数据结构

### 一、图解数据结构和算法-网站

红黑树是一种常见的自平衡二叉查找树，常用于关联数组、字典，在各种语言的底层实现中被广泛应用，Java 的 TreeMap 和 TreeSet 就是基于红黑树实现的。

本篇分享将为读者讲解红黑树的定义、创建和用途。

### 20.5.3 红黑树的定义

1. 每个节点或者是黑色，或者是红色。
2. 根节点是黑色。
3. 每个叶子节点是黑色。
4. 如果一个节点是红色的，则它的子节点必须是黑色的
5. 从任意一个节点到叶子节点，经过的黑色节点是一样的。

这段关于 红黑树 的描述来源于 《算法导论》，你看完这段话可能一脸懵逼。

本文希望能够由浅入深地、渐进式地引导读者了解红黑树，因此我们会先从红黑树的意义说起，为什么我们需要一棵红黑树。

### 20.5.4 平衡二叉查找树

我们以这样一个数组为例 [42, 37, 18, 12, 11, 6, 5] 构建一棵二叉搜索树，由于数组中任意一点都可以作为二叉搜索树的根节点，因此这棵二叉搜索树并不唯一，我们来看一个极端的例子（以42作为根节点，顺序插入元素）



在这个例子中，二叉搜索树退化成了链表，搜索的时间复杂度为  $O(n)$ ，失去了作为一棵二叉搜索树的意义。

为了让二叉搜索树不至于太“倾斜”，我们需要构建一棵 **平衡二叉搜索树**。



可以看出，平衡二叉搜索树的搜索时间复杂度为 $O(\log n)$ ，避免了因为随机选取根节点构建二叉搜索树而可能造成的退化成链表的情况。下面再抄一段平衡二叉搜索树的官方定义：

平衡二叉查找树：简称平衡二叉树。是由前苏联的数学家 Adel'se-Velskil 和 Landis 在 1962 年提出的高度平衡的二叉树，根据科学家的英文名也称为 AVL 树。它具有如下几个性质：

性质1. 可以是空树。

性质2 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1

(如果读者还不清楚平衡二叉搜索树的概念，可以点击查阅前文 [动画：\\*\\*什么是平衡二叉树\\*\\*](#)，本文不再详细介绍平衡二叉搜索树)

## 20.5.5树

经过上面的例子，我们可以知道，构建一棵平衡的二叉搜索树的关键在于选取“正确”的根节点，那么我们如何在每次构建平衡二叉搜索树时都能选取合适的根节点呢，这里就要用到另一种重要的树：2-3树（读作二三树），2-3树和红黑树是等价的，理解2-3树对理解红黑树以及B类树都有很大的帮助。

### 2-3树的基本概念

所谓2-3树，即满足二叉搜索树的性质，且节点可以存放一个元素或者两个元素，每个节点有两个或三个孩子的树。

- 性质1：满足二叉搜索树的性质
- 性质2：节点可以存放一个或两个元素
- 性质3：每个节点有两个或三个子节点

2-3树本质上也是一棵搜索树，和二叉搜索树的区别在于，2-3的节点可能存放 2 个元素，而且每个节点可能拥有 3 个子节点。

2-3树存在以下两种节点：2-节点（存在两个子节点）和3-节点（存在3个子节点）



## 2-3树的创建

下面我们来看如何创建一棵2-3树，创建2-3树的规则如下：

规则1. 加入新节点时，不会往空的位置添加节点，而是添加到最后一个叶子节点上

规则2. 四节点可以被分解三个2-节点组成的树，并且分解后新树的根节点需要向上和父节点融合

我们依然使用上面的示例数组[42, 37, 18, 12, 11, 6, 5]，依然使用顺序插入的方式来构建2-3树，看看是否会出现退化成链表的情况。



我们可以注意到，在创建2-3树的每一步中，整棵树始终保持平衡。

既然2-3树已经能够保持自平衡，为什么我们还需要一棵红黑树呢，这是因为 **2-3树这种每个节点储存1~2个元素以及拆分节点向上融合的性质不便于代码操作**，因此我们希望通过一些规则，将2-3树转换成二叉树，且转换后的二叉树依然能保持平衡性。

## 2-3树和红黑树的等价性

本小节我们以一棵2-3树为例，将其从2-3树转换成为一棵红黑树，从而学习了解2-3树和红黑树的转换规则，并体会2-3树和红黑树之间的等价性。

对于2-3树中的2-节点来说，本身就和二叉搜索树的节点无异，可以直接转换为红黑树的一个黑节点，但是对于3-节点来说，我们需要进行一点小转换：

1. 将3-节点拆开，成为一棵树，并且3-节点的左元素作为右元素的子树
2. 将原来的左元素标记为红色（表示红色节点与其父节点在2-3树中曾是平级的关系）



我们来转换一棵复杂点的2-3树，根据上边的两条转换规则，我们将2-节点直接转换为黑色节点，将3-节点拆成一棵子树，并给左元素标上红色，这个过程应该不难理解，另外我们可以注意到，由于红色节点是由3-节点拆分而来，因此所有的红色节点都只会出现在左子树上。



## 20.5.6红黑树的性质和复杂度分析

### 红黑树基本性质分析

在完成了2-3树到红黑树的转换之后，我们重新审视红黑树的五条性质：

(1) 每个节点或者是黑色，或者是红色

这是红黑树的定义，没什么好说的。

(2) 根节点是黑色

根节点要么对应2-3树的2-节点或者3-节点，而这两者的根节点都是黑色的，因而根节点必然是黑色。从上图2-3树节点和红黑树节点对应关系就能很容易看出来

(3) 每个叶子节点是黑色

注意，这里的叶子是指的为空的叶子节点，上图的红黑树的完整形式应该是这样的：



(4) 如果一个节点是红色的，则它的子节点必须是黑色的

由于红黑树的每个节点都由2-3树转换而来，红色节点连接的节点必然是一个2-节点或者3-节点，而无论是2-节点还是3-节点，其根节点都是黑色的，因此红色节点的子节点必然是黑色的

(5) 从任意一个节点到叶子节点，经过的黑色节点是一样多的

这是红黑树最重要的一条性质，也是红黑树的价值所在。由于红黑树是由2-3树转换而来，因此每一个黑色节点必然对应2-3树的某个2-节点或者3-节点，因此红黑树的黑节点也能拥有2-3树的平衡性。

如果对这条性质还不够理解，可以对着上文2-3树和红黑树的转换图再理解理解。

## 红黑树时间复杂度分析

网上有很多使用数学归纳法来计算红黑树时间复杂度的证明了，这里就不再赘述。

我们可以简单思考一下，对于一棵普通的平衡二叉搜索树来说，它的搜索时间复杂度为 $O(\log n)$ ，而作为红黑树，存在着最坏的情况，也就是查找的过程中，经过的节点全都是原来2-3树里的3-节点，导致路径延长两倍，时间复杂度为 $O(2\log n)$ ，由于时间复杂度的计算可以忽略系数，因此红黑树的搜索时间复杂度依然是 $O(\log n)$ ，当然，由于这个系数的存在，在实际使用中，红黑树会比普通的平衡二叉树（AVL树）搜索效率要低一些。



既然红黑树的效率比AVL树的效率低，那么红黑树的优势体现在哪呢？

事实上，红黑树的优势体现在它的插入和删除操作上，红黑树的插入和删除相较于AVL树的性能会高一些，在后续红黑树的创建章节中，读者应该能够体会到红黑树在调整树平衡操作上的优势。

## 20.5.7 红黑树的创建

上文中我们讲解了如何由2-3树转换一棵红黑树，下面我们就来看看如何不经过2-3树直接创建一棵红黑树，毕竟我们写代码的时候不能先创建一棵2-3树再转化成红黑树吧。

我们回想一下2-3树的创建规则：

规则1. 加入新节点时，不会往空的位置添加节点，而是添加到最后一个叶子节点上

规则2. 四节点可以被分解三个2-节点组成的树，并且分解后新树的根节点需要向上和父节点融合

简单来说，2-3树的创建分为「融合」和「拆分」两步，为了实现这两步，我们需要在创建二叉树的基础操作上增加另外几个操作，分别是：

1. 保持根节点黑色
2. 左旋转
3. 右旋转
4. 颜色翻转

### 保持根节点黑色和左旋转

由于我们往2-3树插入节点时做的都是融合，因此新加入的节点和原位置的节点是平级关系，所以我们往红黑树里增加节点的时候，增加的都是红色节点。



我们插入第一个红色节点42，哦吼，第一步就与红黑树的性质2「根节点是黑色」冲突，为了解决这种冲突，**我们将根节点变成黑色**。



下面我们继续插入节点37，同样的，新插入的节点都为红色



这里我们要思考一下，如果我们颠倒顺序，先插入37再插入42呢，是直接把42加到37的右子树上么，这显然是错误的，因为在前边2-3树转红黑树的过程中，我们已经了解到，所有的红色节点都只会出现在左子树上，因此我们需要进行**左旋转**，将节点的位置和颜色旋转过来。



上面是两个独立的节点，如果节点拥有左右子树，在旋转后仍然能满足二叉搜索树的性质吗，我们需要推广到一般情形。

我们来看一个例子：





经过以上几步，我们就完成了一般情形下的左旋转，我们可以对应地写几句伪代码，这里把 37 称作 node，42 称作 target

```
function leftRotate(node) { node.right = target.left
target.left = node target.color = node.color
node.color = RED } function flipColors(node) {
node.color = RED node.left.color = BLACK
node.right.color = BLACK } function rightRotate(node) {
node.left = T1 target.right = node target.color
= node.color node.color = RED } function add(node) {
//如果右节点为红色，左节点为黑色，那么进行左旋转，对应情况2
if(isRed(node.right) && !isRed(node.left)) node =
leftRotate(node) //如果左节点为红色，左节点的左节点也为红色，
那么进行右旋转，对应情况3 if(isRed(node.left) &&
isRed(node.left.left)) node = rightRotate(node) //如
果左右节点都为红色，那么进行颜色翻转，对应情况4
if(isRed(node.left) && isRed(node.right))
flipColors(node) }
```

## 20.6 浅谈单链表与双链表的区别

昨天面试官面试的时候问了我一道关于链表的问题：情境如下

面试官：请说一下链表跟数组的区别？

我：数组静态分配内存，链表动态分配内存；数组在内存中连续，链表不连续；数组利用下标定位，时间复杂度为O(1)，链表定位元素时间复杂度O(n)；数组插入或删除元素的时间复杂度O(n)，链表的时间复杂度O(1)。

根据以上分析可得出数组和链表的优缺点如下：

### 数组的优点

- 随机访问性强（通过下标进行快速定位）
- 查找速度快

## 数组的优点

- 插入和删除效率低（插入和删除需要移动数据）
- 可能浪费内存（因为是连续的，所以每次申请数组之前必须规定数组的大小，如果大小不合理，则可能会浪费内存）
- 内存空间要求高，必须有足够的连续内存空间。
- 数组大小固定，不能动态拓展

## 链表的优点

- 插入删除速度快（因为有next指针指向其下一个节点，通过改变指针的指向可以方便的增加删除元素）
- 内存利用率高，不会浪费内存（可以使用内存中细小的不连续空间（大于node节点的大小），并且在需要空间的时候才创建空间）
- 大小没有固定，拓展很灵活。

## 链表的缺点

- 不能随机查找，必须从第一个开始遍历，查找效率低

面试官：那请说一下单链表和双链表的区别？

我：

单链表只有一个指向下一结点的指针，也就是只能next。双链表除了有一个指向下一结点的指针外，还有一个指向前一结点的指针，可以通过prev()快速找到前一结点，顾名思义，单链表只能单向读取

面试官：从你的描述来看，双链表的在查找、删除的时候可以利用二分法的思想去实现，那么这样效率就会大大提高，但是为什么目前市场应用上单链表的应用要比双链表的应用要广泛的多呢？

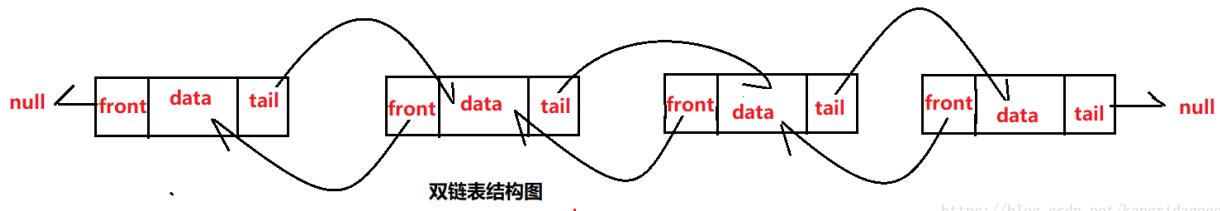
我：.....这个我真的不知道，然后面试官就提醒我从存储效率上来考虑问题.....

我回来后百度了下，发现网上的回答大都是关于链表的代码实现的，并没有关于链表本质深层次的分析，于是我便做以下分析：

单链表与双链表的结构图如下：



<https://blog.csdn.net/kangxidagege>



<https://blog.csdn.net/kangxidagege>

从以上结构可以得出双链表具有以下优点：

- 1、删除单链表中的某个结点时，一定要得到待删除结点的前驱，得到该前驱有两种方法，第一种方法是在定位待删除结点的同时一路保存当前结点的前驱。第二种方法是在定位到待删除结点之后，重新从单链表表头开始来定位前驱。尽管通常会采用方法一。但其实这两种方法的效率是一样的，指针的总的移动操作都会有 $2*i$ 次。而如果用双向链表，则不需要定位前驱结点。因此指针总的移动操作为*i*次。
- 2、查找时也一样，我们可以借用二分法的思路，从head（首节点）向后查找操作和last（尾节点）向前查找操作同步进行，这样双链表的效率可以提高一倍。

可是为什么市场上单链表的使用多余双链表呢？

从存储结构来看，每个双链表的节点要比单链表的节点多一个指针，而长度为n就需要  $n*length$ （这个指针的length在32位系统中是4字节，在64位系统中是8个字节）的空间，这在一些追求时间效率不高应用下并不适应，因为它占用空间大于单链表所占用的空间；这时设计者就会采用以时间换空间的做法，这时一种工程总体上的衡量。

## 第八章 新技术篇

# 第一节 实战问题篇

## 8.1 Android工程中方法数超过65536解决方法

### 8.1.1 概况

项目中由于平时使用了不少的三方库，同时随着App的不断迭代自身代码也在不断增加，这时候方法数超过了65536。带来的后果就是编译和打包失败。

### 8.1.2 原因

当方法数超过65536的时候，会报如下的错误。

```
The number of method references in a .dex file cannot exceed 64K. Learn how to resolve this issue at https://developer.android.com/tools/building/multidex.html
1 com.android.dex.DexIndexOverflowException: method ID not in [0, 0xffff]: 65536
```

65536它代表了引用的总数上限，单个Dex文件（Dalvik可执行文件）中最多可以包含65536个方法；

#### 2.1 关于64 k引用限制

Android应用程序（APK）在Dalvik可执行文件的形式包含可执行的字节码文件（DEX）文件，其中包含已编译的代码（运行你的应用程序）。 Dex文件最多能包含65536个方法：包括Android框架方法、library方法的总数、和你自己的代码方法总数。因为65536等于 $64 \times 1024$ ，这一限制被称为“64k引用限制”。

这个极限就要求我们配置应用程序的构建过程，需要生成多个DEX文件，所以称为multidex 配置。

#### 2.2 不同版本下虚拟机

Android 5.0 (API level 21) 之前的系统使用 Dalvik 虚拟机。默认情况下，Dalvik 限制一个apk只有一个Dex文件。为了绕过这个限制，我们可以使用com.android.support:multidex也就是Android官方为我们提供的解决这个问题的一个库，它成为我们APK的主要Dex文件的一部分，负责管

理我们APK访问其他Dex文件和代码。 (注意: 如果咱的项目 minSdkVersion是20或更低, 运行到Android 4.4(API leve 20)或者更低版本的设备上时需要禁用AndroidStudio的即时运行)

Android 5.0 (API leve 21) 和更高的系统使用 runtime是ART , 原生支持从应用的apk文件加载多个DEX文件。ART在安装应用时预编译应用程序, 会扫描多个classes(..N).dex文件编译成一个.oat的文件。

由于这里只是为了阐明“64 k引用限制”的历史背景, 所以关于两种虚拟机的知识点这里不做展开。

### 8.1.3解决方法

#### 尽量避免64k限制

具体建议:

- 1.选择三方依赖库的时候做好调研工作, 选择满足自己功能的并且尽量小的。 (建议能用源码的用源码, 对于以后自己的定制和维护很有好处)
- 2.正式打包构建的时候, 使用代码混淆器ProGuard混淆移除未使用的代码, 也就是不把没有使用的代码打包到我们的apk中。

#### 解决64k问题

**app中build.gradle中添加** `multidexEnabled true` **和** `compile 'com.android.support:multidex:1.0.1'` **;含义: 支持多个dex和依赖multidex库。**

```
apply plugin: 'com.android.application' android { ---省略代码--- compileSdkVersion 23 buildToolsVersion '26.0.2' defaultConfig { minSdkVersion 17 targetSdkVersion 22 versionCode 61 versionName "4.2.5" // Enabling multidex support. multidexEnabled true } ---省略代码-- }
```

```
-}dependencies { ---省略代码--- compile 'com.android.support:multidex:1.0.1'}
```

#### 继承`android.support.multidex.MultiDexApplication`类

分两种情况：

第一种有自己的Application

重写Applicatio中这个方法如下

```
 @Override protected void attachBaseContext(Context base) { super.attachBaseContext(base); MultiDex.install(this); }
```

第二种没有自己的Application

如果我们的APP没有重写过Application类，我们直接继承

MultiDexApplication，然后在manifest.xml中注册Application即可。

与此同时可能会报 `java.lang.OutOfMemoryError: Java heap space` 错误，那么解决方法是：在app的build.gradle中android中添加如下代码

```
//加大java堆内存大小 dexOptions { javaMaxHeapSize "2g"//这里2g或者4g都可以 }
```

### 8.1.4 multidex库的一些限制因素

#### 第二个Dex文件过大问题

1.Dex文件安装到设备的过程非常复杂，如果第二个Dex文件太大，可能导致应用无响应。此时应该使用ProGuard减小Dex文件的大小。

#### 系统低版本问题

1.由于Dalvik linearAlloc的Bug，应用可能无法在Android 4.0之前的版本启动，如果你的应用要支持这些版本就要多执行测试。

2.同样因为Dalvik linearAlloc的限制，如果请求大量内存可能导致崩溃。

Dalvik linearAlloc是一个固定大小的缓冲区。在应用的安装过程中，系统会运行一个名为dexopt的程序为该应用在当前机型中运行做准备。

dexopt使用LinearAlloc来存储应用的方法信息。Android 2.2和2.3的缓冲区只有5MB，Android 4.x提高到了8MB或16MB。当方法数量过多导致超出缓冲区大小时，会造成dexopt崩溃。

3.Multidex构建工具还不支持指定哪些类必须包含在首个Dex文件中，因此可能会导致某些类库（例如某个类库需要从原生代码访问Java代码）无法使用。

总之就是在做低版本兼容的时候可能会崩溃，要多测试。（不过现在基本面向的用户使用的都是4.4以上的系统，这些情况出现的几率大大降低）。

## 第九章 面试篇

### 第一节 开源文档

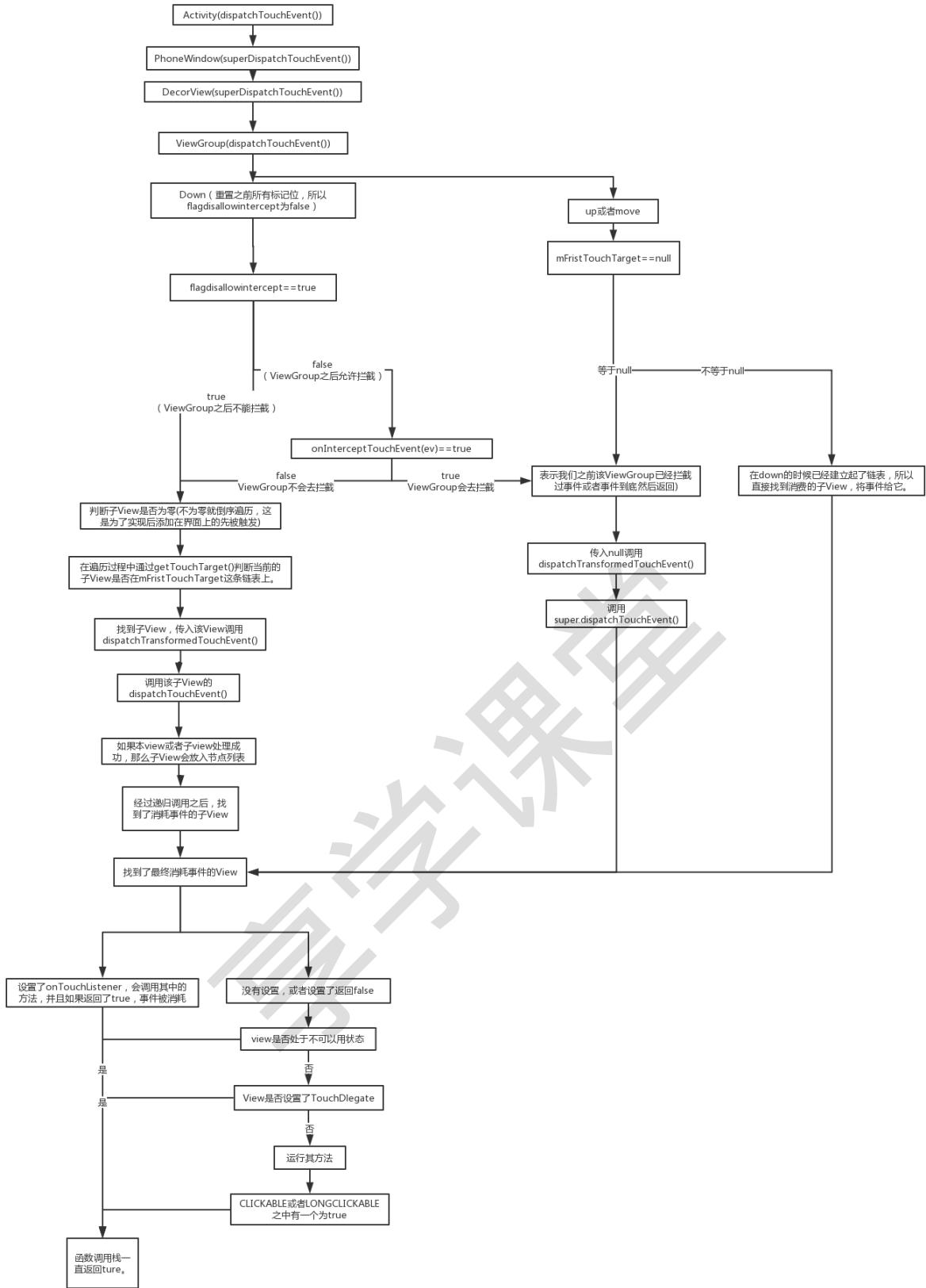
- Awesome-Android-Interview 随着Android技术发展的成熟，Kotlin、大前端技术Flutter、RN、小程序等一下子就进入了我们的视野内，同时，Android自身的技术栈也正在不断扩展，比如在国外大热的Jetpack。因此，Android开发者们越来越焦虑，越来越迷茫，每个人的时间和精力是有限的，我们到底应该学什么才能有效地提高自身的竞争力呢？其实，首先我们应该优先深入学习工作中用到的技术，其次，关注这2年来Android最新的面试题所涉及的知识点，根据自身的实际情况有选择地进行针对性的学习和提升。只有这样，自身才不会被所谓的互联网寒冬吓倒。Awesome-Android-Interview搜集了国内一线及二线互联网公司最常出现的面试题，非常全面，笔者花费了很大的精力和时间，希望得到大家的支持。
- AndroidGuide 这是一份Android开发从基础入门到进阶的完整（并不是）指南，所有文章都是本人这几年时间里一字一字码出来的，文章的更新方向和更新频率以我的学习计划为导向，会一直持续密集更新下去...
- LearningNotes
- Android和Flutter笔记文档 基于开源文档Mkdocs部署搭建的学习笔记个人网站
- BAT大厂Android面试知识点，请客官拿好 年年寒冬，年年也挡不住一个安卓程序员追求大厂的决心。想要进入大厂，我们需要掌握哪些知识点呢？这里，我为大家梳理了一个整体的知识架构。整体包括Java、Android、算法、网络等，并且我也在相应知识点下推荐了与该知识点相关的书籍与博客。希望大家阅读之后，能帮助大家完善与整理自己的知识体系。祝大家早日进入自己理想的公司~~

## 第二节 面试题合集

如今安卓开发不像前几年那么热门，但是高级人才依然紧缺，大家看着这句话是不是很熟悉，因为 web 高级人才也紧缺，c++ 高级人才一样紧缺，那么到了人工智能时代，人工智能时代的高级人才也同样会紧缺！似乎是高级人才的人在其他领域也是高级人才，而不是因为选择了热门才会一帆风顺。

**网上高级工程师面试相关文章鱼龙混杂，要么一堆内容，要么内容质量太浅，** 鉴于此我整理了如下安卓开发高级工程师面试题以及答案帮助大家顺利进阶为高级工程师，目前我就职于某大厂安卓高级工程师职位，在当下大环境下也想为安卓工程师出一份力，通过我的技术经验整理了面试经常问的题，答案部分会是一篇文章或者几篇文章，都是我认真看过并且觉得不错才整理出来，大家知道高级工程师不会像刚入门那样被问的问题一句话两句话就能表述清楚，所以我通过过滤好文章来帮助大家理解，进入正题

### 2.1 android事件分发机制，请详细说下整个流程



<http://blog.csdn.net/>

## 2.2 android view绘制机制和加载过程，请详细说下整个流程

- 1. ViewRootImpl会调用performTraversals(), 其内部会调用performMeasure()、performLayout、performDraw()。

- 2.performMeasure()会调用最外层的ViewGroup的measure()->onMeasure()。  
ViewGroup的onMeasure()是抽象方法，但其提供了measureChildren()，这之中会遍历子View然后循环调用measureChild()  
这之中会用getChildMeasureSpec() + 父View的MeasureSpec + 子View的LayoutParams一起获取本View的MeasureSpec，  
然后调用子View的measure()到View的onMeasure()->setMeasureDimension(getDefaultSize(), getDefaultSize()), getDefaultSize()默认返回measureSpec的测量数值，所以继承View进行自定义的wrap\_content需要重写。
- 3.performLayout()会调用最外层的ViewGroup的layout(l,t,r,b),本View在其中使用setFrame()设置本View的四个顶点位置。  
在onLayout(抽象方法)中确定子View的位置，如LinearLayout会遍历子View，循环调用setChildFrame()->子View.layout()。
- 4.performDraw()会调用最外层ViewGroup的draw():  
其中会先后调用background.draw()(绘制背景)、onDraw()(绘制自己)、dispatchDraw()(绘制子View)、onDrawScrollBars()(绘制装饰)。
- 5.MeasureSpec由2位SpecMode(UNSPECIFIED、EXACTLY(对应精确值和match\_parent)、AT\_MOST(对应wrap\_content))和30位SpecSize组成一个int,DecorView的MeasureSpec由窗口大小和其LayoutParams决定，其他View由父View的MeasureSpec和本View的LayoutParams决定。  
ViewGroup中有getChildMeasureSpec()来获取子View的MeasureSpec。
- 6.三种方式获取measure()后的宽高：
  - 1.Activity#onWindowFocusChange()中调用获取
  - 2.view.post(Runnable)将获取的代码投递到消息队列的尾部。
  - 3.ViewTreeObservable.

## 2.3 android四大组件的加载过程，请详细介绍下

1.android四大组件的加载过程:[请看这篇博客](#)

## 2.4 Activity的启动模式

- 1.standard: 默认标准模式，每启动一个都会创建一个实例，
- 2.singleTop: 栈顶复用，如果在栈顶就调用onNewIntent复用，从onResume()开始
- 3.singleTask: 栈内复用，本栈内只要用该类型Activity就会将其顶部的activity出栈
- 4.singleInstance: 单例模式，除了3中特性，系统会单独给该Activity创建一个栈。

## 2.5 A、B、C、D分别是四种Activity的启动模式，那么A->B->C->D->A->B->C->D分别启动，最后的activity栈是怎样的

- 1.这个题目需要深入了解activity的启动模式
- 2.最后的答案是：两个栈，前台栈是只有D，后台栈从底至上是A、B、C 分前台栈和后台栈

## 2.6 Activity缓存方法

1.配置改变导致Activity被杀死，横屏变竖屏：在onStop之前会调用onSaveInstanceState()保存数据在重建Activity之后，会在onStart()之后调用onRestoreInstanceState(),并把保存下来的Bundle传给onCreate()和它会默认重建Activity当前的视图，我们可以在onCreate()中，回复自己的数据。

2.内存不足杀掉Activity，优先级分别是：前台可见，可见非前台，后台。

## 2.7 Service的生命周期，两种启动方法，有什么区别

- 1.context.startService() ->onCreate()->onStart()->Service running->(如果调用context.stopService() )->onDestroy() ->Service shut down
  - 1.如果Service还没有运行，则调用onCreate()然后调用onStart();

- 2.如果Service已经运行，则只调用onStart()，所以一个Service的**onStart方法可能会重复调用多次**。
  - 3.调用stopService的时候直接onDestroy，
  - 4.如果是调用者自己直接退出而没有调用stopService的话，Service会一直在后台运行。该Service的调用者再启动起来后可以通过stopService关闭Service。
- 2.context.bindService()->onCreate()->onBind()->Service running->onUnbind() -> onDestroy() ->Service stop
    - 1.onBind将返回给客户端一个IBind接口实例，IBind允许客户端回调服务的方法，比如得到Service运行的状态或其他操作。
    - 2.这个时候会把调用者和Service绑定在一起，Context退出了，Service就会调用onUnbind->onDestroy相应退出。
    - 3.所以调用bindService的生命周期为：onCreate -> onBind(**只一次，不可多次绑定**) -> onUnbind -> onDestroy。

## 2.8怎么保证service不被杀死

- 1.提升service优先级
- 2.提升service进程优先级
- 3.onDestroy方法里重启service

## 2.9静态的Broadcast 和动态的有什么区别

- 1.动态的比静态的安全
- 2.静态在app启动的时候就初始化了 动态使用代码初始化
- 3.静态需要配置 动态不需要
- 4.生存期，静态广播的生存期可以比动态广播的长很多
- 5.优先级动态广播的优先级比静态广播高

## 2.10Intent可以传递哪些数据类型

- 1.Serializable
- 2.charsequence: 主要用来传递String, char等
- 3.parcelable
- 4.Bundle

## 2.11Json有什么优劣势、解析的原理

- 1.JSON的速度要远远快于XML
- 2.JSON相对于XML来讲，数据的体积小
- 3.JSON对数据的描述性比XML较差
- 4.解析的基本原理是：词法分析

## 2.12一个语言的编译过程

- 1.词法分析：将一串文本按规则分割成最小的结构，关键字、标识符、运算符、界符和常量等。一般实现方法是自动机和正则表达式
- 2.语法分析：将一系列单词组合成语法树。一般实现方法有自顶向下和自底向上
- 3.语义分析：对结构上正确的源程序进行上下文有关性质的审查
- 4.目标代码生成
- 5.代码优化：优化生成的目标代码，

## 2.13动画有哪几类，各有什么特点

- 1.动画的基本原理：其实就是利用插值器和估值器，来计算出各个时刻View的属性，然后通过改变View的属性来，实现View的动画效果。
- 2.View动画：只是影像变化，view的实际位置还在原来的地方。
- 3.帧动画是在xml中定义好一系列图片之后，使用AnimationDrawable来播放的动画。
- 4.View的属性动画：
  - 1.插值器 (Interpolator)：作用是根据时间的流逝的百分比来计算属性改变的百分比
  - 2.估值器 (TypeEvaluator)：在1的基础上由这个东西来计算出属性到底变化了多少数值的类

## 2.14Handler、Looper消息队列模型，各部分的作用

- 1.MessageQueue：读取会自动删除消息，单链表维护，在插入和删除上有优势。在其next()中会无限循环，不断判断是否有消息，有就返回这条消息并移除。

- 2Looper：Looper创建的时候会创建一个MessageQueue，调用loop()方法的时候消息循环开始，loop()也是一个死循环，会不断调用messageQueue的next()，当有消息就处理，否则阻塞在messageQueue的next()中。当Looper的quit()被调用的时候会调用messageQueue的quit()，此时next()会返回null，然后loop()方法也跟着退出。
- 3Handler：在主线程构造一个Handler，然后在其他线程调用sendMessage()，此时主线程的MessageQueue中会插入一条message，然后被Looper使用。
- 4系统的主线程在ActivityThread的main()为入口开启主线程，其中定义了内部类Activity.H定义了一系列消息类型，包含四大组件的启动停止。
- 5.MessageQueue和Looper是一对一关系，Handler和Looper是多对一

## 2.15怎样退出终止App

1.自己设置一个Activity的栈，然后一个个finish()

## 2.16Android IPC:Binder原理

- 1.在Activity和服务进行通讯的时候，用到了Binder。
  - 1.当属于同一个进程我们可以继承Binder然后在Activity中对Service进行操作
  - 2.当不属于同一个进程，那么要用到AIDL让系统给我们创建一个Binder，然后在Activity中对远端的Service进行操作。
- 2.系统给我们生成的Binder：
  - 1.Stub类中有：接口方法的id，有该Binder的标识，有asInterface(IBinder)（让我们在Activity中获取实现了Binder的接口，接口的实现都在Service里，同进程时候返回Stub，不同进程则返回Proxy），有onTransact()这个方法是在不同进程的情况下，Proxy通过这个方法，在Activity进行远端调用实现Activity操作Service
  - 2.Proxy类是代理，在Activity端，其中有：IBinder mRemote（这就是远端的Binder），两个接口的实现方法不过是代理最终还是要在远端

的onTransact()中进行实际操作。

- 3.哪一端的Binder是副本，该端就可以被另一端进行操作，因为Binder本体在定义的时候可以操作本端的东西。所以可以在Activity端传入本端的Binder，让Service端对其进行操作称为Listener，可以用**RemoteCallbackList**这个容器来装Listener，防止Listener因为经历过序列化而产生的问题。
- 4.当Activity端向远端进行调用的时候，当前线程会挂起，当方法处理完毕才会唤醒。
- 5.如果一个AIDL就用一个Service太奢侈，所以可以使用Binder池的方式，建立一个AIDL其中的方法是返回IBinder，然后根据方法中传入的参数返回具体的AIDL。
- 6.IPC的方式有：Bundle（在Intent启动的时候传入，不过是一次性的），文件共享（对于SharedPreference是特例，因为其在内存中会有缓存），使用Messenger（其底层用的也是AIDL，同理要操作哪端，就在哪端定义Messenger），AIDL，ContentProvider（在本进程中继承实现一个ContentProvider，在增删改查方法中调用本进程的SQLite，在其他进程中查询），Socket

## 17.描述一次跨进程通讯

- 1.client、proxy、serviceManager、BinderDriver、impl、service
- 2.client发起一个请求service信息的Binder请求到BinderDriver中，serviceManager发现BinderDiriver中有自己的请求 然后将clinet请求的service的数据返回给client这样完成了一次Binder通讯
- 3.clinet获取的service信息就是该service的proxy，此时调用proxy的方法，proxy将请求发送到BinderDriver中，此时service的 Binder线程池循环发现有自己的请求，然后用impl就处理这个请求最后返回，这样完成了第二次Binder通讯
- 4.中间client可挂起，也可以不挂起，有一个关键字oneway可以解决这个

## 18.android重要术语解释

- 1.ActivityManagerServices，简称AMS，服务端对象，负责系统中所有Activity的生命周期

- 2.ActivityThread，App的真正入口。当开启App之后，会调用main()开始运行，开启消息循环队列，这就是传说中的UI线程或者叫主线程。与ActivityManagerServices配合，一起完成Activity的管理工作
- 3.ApplicationThread，用来实现ActivityManagerService与ActivityThread之间的交互。在ActivityManagerService需要管理相关Application中的Activity的生命周期时，通过ApplicationThread的代理对象与ActivityThread通讯。
- 4.ApplicationThreadProxy，是ApplicationThread在服务器端的代理，负责和客户端的ApplicationThread通讯。AMS就是通过该代理与ActivityThread进行通信的。
- 5.Instrumentation，每一个应用程序只有一个Instrumentation对象，每个Activity内都有一个对该对象的引用。Instrumentation可以理解为应用进程的管家，ActivityThread要创建或暂停某个Activity时，都需要通过Instrumentation来进行具体的操作。
- 6.ActivityStack，Activity在AMS的栈管理，用来记录已经启动的Activity的先后关系，状态信息等。通过ActivityStack决定是否需要启动新的进程。
- 7.ActivityRecord，ActivityStack的管理对象，每个Activity在AMS对应一个ActivityRecord，来记录Activity的状态以及其他管理信息。其实就是服务器端的Activity对象的映像。
- 8.TaskRecord，AMS抽象出来的一个“任务”的概念，是记录ActivityRecord的栈，一个“Task”包含若干个ActivityRecord。AMS用TaskRecord确保Activity启动和退出的顺序。如果你清楚Activity的4种launchMode，那么对这个概念应该不陌生。

## 2.17理解Window和WindowManager

- 1.Window用于显示View和接收各种事件，Window有三种类型：应用Window(每个Activity对应一个Window)、子Window(不能单独存在，附属于特定Window)、系统window(Toast和状态栏)
- 2.Window分层级，应用Window在1-99、子Window在1000-1999、系统Window在2000-2999.WindowManager提供了增删改View三个功能。

- 3.Window是个抽象概念：每一个Window对应着一个View和ViewRootImpl，Window通过ViewRootImpl来和View建立联系，View是Window存在的实体，只能通过WindowManager来访问Window。
- 4.WindowManager的实现是WindowManagerImpl其再委托给 **WindowManagerGlobal**来对Window进行操作，其中有四个List分别储存对应的View、ViewRootImpl、WindowManger.LayoutParams和正在被删除的View
- 5.Window的实体是存在于远端的WindowMangerService中，所以增删改Window在本端是修改上面的几个List然后通过ViewRootImpl重绘View，通过WindowSession(每个应用一个)在远端修改Window。
- 6.Activity创建Window：Activity会在attach()中创建Window并设置其回调(onAttachedToWindow()、dispatchTouchEvent()),Activity的Window是由Policy类创建PhoneWindow实现的。然后通过Activity#setContentView()调用PhoneWindow的setContentView。

## 2.18 Bitmap的处理

- 1.当使用ImageView的时候，可能图片的像素大于ImageView，此时就可以通过BitmapFactory.Option来对图片进行压缩，inSampleSize表示缩小 $2^{(inSampleSize-1)}$ 倍。
- 2.Bitmap的缓存：
  - 1.使用LruCache进行内存缓存。
  - 2.使用DiskLruCache进行硬盘缓存。
- 3.实现一个ImageLoader的流程：同步异步加载、图片压缩、内存硬盘缓存、网络拉取
  - 1.同步加载只创建一个线程然后按照顺序进行图片加载
  - 2.异步加载使用线程池，让存在的加载任务都处于不同线程
  - 3.为了不开启过多的异步任务，只在列表静止的时候开启图片加载

## 2.19 如何实现一个网络框架(参考Volley)

- 1.缓存队列,以url为key缓存内容可以参考Bitmap的处理方式，这里单独开启一个线程。
- 2.网络请求队列，使用线程池进行请求。

- 3.提供各种不同类型的返回值的解析如String, Json, 图片等等。

## 2.20ClassLoader的基础知识

- 1.双亲委托：一个ClassLoader类负责加载这个类所涉及的所有类，在加载的时候会判断该类是否已经被加载过，然后会递归去他父ClassLoader中找。
- 2.可以动态加载Jar通过URLClassLoader
- 3.ClassLoader 隔离问题 JVM识别一个类是由：ClassLoader id+PackageName+ClassName。
- 4.加载不同Jar包中的公共类：
  - 1.让父ClassLoader加载公共的Jar，子ClassLoader加载包含公共Jar的Jar，此时子ClassLoader在加载公共Jar的时候会先去父ClassLoader中找。(只适用Java)
  - 2.重写加载包含公共Jar的Jar的ClassLoader，在loadClass中找到已经加载过公共Jar的ClassLoader，也就是把父ClassLoader替换掉。(只适用Java)
  - 3.在生成包含公共Jar的Jar时候把公共Jar去掉。

## 2.21插件化框架描述：dynamicLoadApk为例子

- 1.可以通过DexClassLoader来对apk中的dex包进行加载访问
- 2.如何加载资源是个很大的问题，因为宿主程序中并没有apk中的资源，所以调用R资源会报错，所以这里使用了Activity中的实现ContextImpl的getAssets()和getResources()再加上反射来实现。
- 3.由于系统启动Activity有很多初始化动作要做，而我们手动反射很难完成，所以可以采用接口机制，将Activity的大部分生命周期提取成接口，然后通过**代理Activity**去调用插件Activity的生命周期。同时如果想增加一个新生命周期方法的时候，只需要在接口中和代理中声明一下就行。
- 4.缺点：
  - 1.慎用this，因为在apk中使用this并不代表宿主中的activity，当然如果this只是表示自己的接口还是可以的。除此之外可以使用that代

替this。

- 2.不支持Service和静态注册的Broadcast
- 3.不支持LaunchMode和Apk中Activity的隐式调用。

## 2.22热修复：Andfix为例子

- 1.大致原理：**apkpatch**（阿里Andfix提供的工具包）将两个apk做一次对比，然后找出不同的部分。可以看到生成的apatch了文件，后缀改成zip再解压开，里面有一个dex文件。  
通过jadex查看一下源码，里面就是被修复的代码所在的类文件，这些更改过的类都加上了一个\_CF的后缀，并且变动的方法都被加上了一个叫@MethodReplace的annotation，通过clazz和method指定了需要替换的方法。然后客户端sdk得到补丁文件后就会根据annotation来寻找需要替换的方法。最后由JNI层完成方法的替换。
- 2.无法添加新类和新的字段、补丁文件很容易被反编译、加固平台可能会使热补丁功能失效

## 2.23线程同步的问题，常用的线程同步

- 1.sycn：保证了原子性、可见性、有序性
- 2.锁：保证了原子性、可见性、有序性
  - 1.自旋锁：可以使线程在没有取得锁的时候，不被挂起，而转去执行一个空循环。
    - 1.优点：线程被挂起的几率减少，线程执行的连贯性加强。用于对于锁竞争不是很激烈，锁占用时间很短的并发线程。
    - 2.缺点：过多浪费CPU时间，有一个线程连续两次试图获得自旋锁引起死锁
  - 2.阻塞锁：没得到锁的线程等待或者挂起，Sycn、Lock
  - 3.可重入锁：一个线程可多次获取该锁，Sycn、Lock
  - 4.悲观锁：每次去拿数据的时候都认为别人会修改，所以会阻塞全部其他线程 Sycn、Lock
  - 5.乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。cas

- 6.显示锁和内置锁:显示锁用Lock来定义、内置锁用synchronized。
- 7.读-写锁:为了提高性能, Java提供了读
- 3.volatle
  - 1.只能保证可见性, 不能保证原子性
  - 2.自增操作有三步, 此时多线程写会出现问题
- 4.CAS (Compare and swap)
  - 1.操作:内存值V、旧的预期值A、要修改的值B, 当且仅当预期值A和内存值V相同时, 将内存值修改为B并返回true, 否则什么都不做并返回false。
  - 2.解释:本地副本为A, 共享内存为V, 线程A要把V修改成B。某个时刻线程A要把V修改成B, 如果A和V不同那么就表示有其他线程在修改V, 此时就表示修改失败, 否则表示没有其他线程修改, 那么把V改成B。
  - 3.局限:如果V被修改成V1然后又被改成V, 此时cas识别不出变化, 还是认为没有其他线程在修改V, 此时就会有问题
  - 4.局限解决:将V带上版本。
- 5.线程不安全到底是怎么回事:
  - 1.一个线程写, 多个线程读的时候, 会造成写了一半就去读
  - 2.多线程写, 会造成脏数据

## 2.24Asynctask和线程池, GC相关 (怎么判断哪些内存该GC, GC算法)

- 1.AsyncTask: 异步任务类, 单线程线程池+Handler
- 2.线程池:
  - 1.ThreadPoolExecutor: 通过Executors可以构造单线程池、固定数目线程池、不固定数目线程池。
  - 2.ScheduledThreadPoolExecutor: 可以延时调用线程或者延时重调度线程。
- 3.GC相关: 重要
  - 1.搜索算法:
    - 1.引用计数

- 2.图搜索，可达性分析
- 2.回收算法：
  - 1.标记清除复制：用于青年代
  - 2.标记整理：用于老年代
- 3.堆分区：
  - 1.青年区eden 80%、survivor1 10%、survivor2 10%
  - 2.老年区
- 4.虚拟机栈分区：
  - 1.局部变量表
  - 2.操作数栈
  - 3.动态链接
  - 4.方法返回地址
- 5.GC Roots:
  - 1.虚拟机栈(栈帧中的本地变量表)中的引用的对象
  - 2.方法区中的类静态属性引用的对象
  - 3.方法区中的常量引用的对象
  - 4.本地方法栈中JNI的引用的对象

## 2.25 网络

- 1.ARPA协议：在IP以太网中，当一个上层协议要发包时，有了该节点的IP地址，ARP就能提供该节点的MAC地址。
- 2.HTTP HTTPS的区别：
  - 1.HTTPS使用TLS(SSL)进行加密
  - 2.HTTPS缺省工作在TCP协议443端口
  - 3.它的工作流程一般如以下方式：
    - 1.完成TCP三次同步握手
    - 2.客户端验证服务器数字证书，通过，进入步骤3
    - 3.DH算法协商对称加密算法的密钥、hash算法的密钥
    - 4.SSL安全加密隧道协商完成
    - 5.网页以加密的方式传输，用协商的对称加密算法和密钥加密，保证数据机密性；用协商的hash算法进行数据完整性保护，保证

## 数据不被篡改

- 3.http请求包结构, http返回码的分类, 400和500的区别
  - 1.包结构:
    - 1.请求: 请求行、头部、数据
    - 2.返回: 状态行、头部、数据
  - 2.http返回码分类: 1到5分别是, 消息、成功、重定向、客户端错误、服务端错误
- 4.Tcp
  - 1.可靠连接, 三次握手, 四次挥手
    - 1.三次握手: 防止了服务器端的一直等待而浪费资源, 例如只是两次握手, 如果s确认之后c就掉线了, 那么s就会浪费资源
      - 1.syn-c = x, 表示这消息是x序号
      - 2.ack-s = x + 1, 表示syn-c这个消息接收成功。syn-s = y, 表示这消息是y序号。
      - 3.ack-c = y + 1, 表示syn-s这条消息接收成功
    - 2.四次挥手: TCP是全双工模式
      - 1.fin-c = x, 表示现在需要关闭c到s了。ack-c = y, 表示上一条s的消息已经接收完毕
      - 2.ack-s = x + 1, 表示需要关闭的fin-c消息已经接收到了, 同意关闭
      - 3.fin-s = y + 1, 表示s已经准备好关闭了, 就等c的最后一条命令
      - 4.ack-c = y + 1, 表示c已经关闭, 让s也关闭
    - 3.滑动窗口, 停止等待、后退N、选择重传
    - 4.拥塞控制, 慢启动、拥塞避免、加速递减、快重传快恢复

## 2.26数据库性能优化：索引和事务，需要找本专门的书大概了解一下

## 2.27APK打包流程和其内容

### 1.流程

- 1.aapt生成R文件

- 2.aidl生成java文件
- 3.将全部java文件编译成class文件
- 4.将全部class文件和第三方包合并成dex文件
- 5.将资源、so文件、dex文件整合成apk
- 6.apk签名
- 7.apk字节对齐

2.内容： so、 dex、 asset、 资源文件

## 2.28网络劫持的类型原理：可以百度一下了解一下具体概念

- 1.DNS劫持、欺骗、污染
- 2.http劫持：重定向、注入js，http注入、报文扩展

1.java类加载过程：

- 1.加载时机：创建实例、访问静态变量或方法、反射、加载子类之前
- 2.验证：验证文件格式、元数据、字节码、符号引用的正确性
- 3.加载：根据全类名获取文件字节流、将字节流转化为静态储存结构放入方法区、生成class对象
- 4.准备：在堆上为静态变量划分内存
- 5.解析：将常量池中的符号引用转换为直接引用
- 6.初始化：初始化静态变量
- 7.书籍推荐：深入理解java虚拟机，博客推荐：[Java/Android阿里面试JVM部分理解](#)

## 32.retrofit的了解

- 1.动态代理创建一个接口的代理类
- 2.通过反射解析每个接口的注解、入参构造http请求
- 3.获取到返回的http请求，使用Adapter解析成需要的返回值。

## 2.29bundle的数据结构，如何存储

- 1.键值对储存
- 2.传递的数据可以是boolean、byte、int、long、float、double、string等基本类型或它们对应的数组，也可以是对象或对象数组。
- 3.当Bundle传递的是对象或对象数组时，必须实现Serializable 或 Parcelable接口

## 2.30listview内点击button并移动的事件流完整拦截过程：

1.点下按钮的时候：

- 1.产生了一个down事件，activity->phoneWindow->ViewGroup->ListView->button,中间如果有重写了拦截方法，则事件被该view拦截可能消耗。
- 2.没拦截，事件到达了button，这个过程中建立了一条事件传递的view链表
- 3.到button的dispatch方法->onTouch->view是否可用->Touch代理

2.移动点击按钮的时候：

- 1.产生move事件，listView中会对move事件做拦截
- 2.此时listView会将该滑动事件消费掉
- 3.后续的滑动事件都会被listView消费掉

3.手指抬起来时候：前面建立了一个view链表，listView的父view在获取事件的时候，会直接取链表中的listView让其进行事件消耗。

## 2.31android的IPC通信方式，线程（进程间）通信机制有哪些

- 1.ipc通信方式：binder、contentprovider、socket
- 2.操作系统进程通讯方式：共享内存、socket、管道

## 2.32操作系统进程和线程的区别

- 1.简而言之,一个程序至少有一个进程,一个进程至少有一个线程.
- 2.线程的划分尺度小于进程, 使得多线程程序的并发性高。
- 3.另外, 进程在执行过程中拥有独立的内存单元, 而多个线程共享内存, 从而极大地提高了程序的运行效率。
- 4.多线程的意义在于一个应用程序中, 有多个执行部分可以同时执行。有将多个线程看做多个独立的应用, 来实现进程的调度和管理以及资源分配

## 2.34HashMap的实现过程:

Capacity就是buckets的数目, Load factor就是buckets填满程度的最大比例。如果对迭代性能要求很高的话不要把capacity设置过大, 也不要将load factor设置过小。

- 1.简单来说HashMap就是一个会自动扩容的数组链表
- 2.put过程
  - 1.对key的hashCode()做hash, 然后再计算index;
  - 2.如果没碰撞直接放到bucket里;
  - 3.如果碰撞了, 以链表的形式存在buckets后;
  - 4.如果碰撞导致链表过长(大于等于TREEIFY\_THRESHOLD), 就把链表转换成红黑树;
  - 5.如果节点已经存在就替换old value(保证key的唯一性)
  - 6.如果bucket满了(超过load factor\*current capacity), 就要resize。
- 3.resize: 当put时, 如果发现目前的bucket占用程度已经超过了Load Factor所希望的比例, 那么就会发生resize。在resize的过程, 简单的说就是把bucket扩充为2倍, 之后重新计算index, 把节点再放到新的bucket中
- 4.get过程
  - 1.根据key的hash算出数组下表
  - 2.使用equals遍历链表进行比较

### **39.mvc、mvp、mvvm:**

- 1.mvc:数据、View、Activity，View将操作反馈给Activity，Activitiy去获取数据，数据通过观察者模式刷新给View。循环依赖
  - 1.Activity重，很难单元测试
  - 2.View和Model耦合严重
- 2.mvp:数据、View、Presenter，View将操作给Presenter，Presenter去获取数据，数据获取好了返回给Presenter，Presenter去刷新View。PV，PM双向依赖
  - 1.接口爆炸
  - 2.Presenter很重
- 3.mvvm:数据、View、ViewModel，View将操作给ViewModel，ViewModel去获取数据，数据和界面绑定了，数据更新界面更新。
  - 1.viewModel的业务逻辑可以单独拿来测试
  - 2.一个view 对应一个 viewModel 业务逻辑可以分离，不会出现全能类
  - 3.数据和界面绑定了，不用写垃圾代码，但是复用起来不舒服

### **2.35java的线程如何实现**

- 1.Thread继承
- 2.Runnale
- 3.Future
- 4.线程池

### **41.ArrayList 如何删除重复的元素或者指定的元素;**

- 1.删除重复： Set
- 2.删除指定：迭代器

### **42.如何设计在 UDP 上层保证 UDP 的可靠性传输;**

- 1.简单来讲，要使用UDP来构建可靠的面向连接的数据传输，就要实现类似于TCP协议的超时重传，有序接受，应答确认，滑动窗口流量控制等机制，等于说要在传输层的上一层（或者直接在应用层）实现TCP协议的可靠数据传输机制。
- 2.比如使用UDP数据包+序列号，UDP数据包+时间戳等方法，在服务器端进行应答确认机制，这样就会保证不可靠的UDP协议进行可靠的数据传输。
- 3.基于udp的可靠传输协议有：RUDP、RTP、UDT

## 2.36Java 中内部类为什么可以访问外部类

1.因为内部类创建的时候，需要外部类的对象，在内部类对象创建的时候会把外部类的引用传递进去

## 2.37设计移动端的联系人存储与查询的功能，要求快速搜索联系人，可以用到哪些数据结构？

数据库索引，平衡二叉树(B树、红黑树)

## 2.38红黑树特点

- 1.root节点和叶子节点是黑色
- 2.红色节点后必须为黑色节点
- 3.从root到叶子每条路径的黑节点数量相同

### 46.linux异步和同步i/o:

- 1.同步：对于client，client一直等待，但是client不挂起：主线程调用
- 2.异步：对于client，client发起请求，service好了再回调client：其他线程调用，调用完成之后进行回调
- 3.阻塞：对于service，在准备io的时候会将service端挂起，直至准备完成然后唤醒service：bio
- 3.非阻塞：对于service，在准备io的时候不会将service端挂起，而是service一直去轮询判断io是否准备完成，准备完成了就进行操作：nio、linux的select、poll、epoll

- 4.多路复用io：非阻塞io的一种优化，java nio，用一个线程去轮询多个io端口是否可用，如果一个可用就通知对应的io请求，这使用一个线程轮询可以大大增强性能。
  - 1.我可以采用 多线程+ 阻塞IO 达到类似的效果，但是由于在多线程 + 阻塞IO 中，每个socket对应一个线程，这样会造成很大的资源占用。
  - 2.而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。
- 5.异步io： aio， 用户线程完全不感知io的进行，所有操作都交给内核，io完成之后内核通知用户线程。
  - 1.这种io才是异步的，2、3、4都是同步io，因为内核进行数据拷贝的过程都会让用户线程阻塞。
  - 2.异步IO是需要操作系统的底层支持，也就是内核支持，Java 7中，提供了Asynchronous IO

## 2.39 ConcurrentHashMap内部实现，HashTable的实现被废弃的原因：

- 1.HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
- 2.ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。 Segment是一种可重入锁ReentrantLock，在 ConcurrentHashMap里扮演锁的角色， HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组， Segment的结构和HashMap类似，是一种数组和链表结构，一个 Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护者一个HashEntry数组里的元素,当对

HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

## 2.40 HandlerThread是什么

1. MessageQueue + Looper + Handler

## 2.41 IntentService是什么

1. 含有HandlerThread的Service，可以多次startService()来多次在子线程中进行onHandleIntent()的调用。

## 2.42 class和dex

- 1. dvm执行的是dex格式文件，jvm执行的是class文件，android程序编译完之后生产class文件。然后dex工具会把class文件处理成dex文件，然后把资源文件和.dex文件等打包成apk文件。
- 2. dvm是基于寄存器的虚拟机，而jvm执行是基于虚拟栈的虚拟机。寄存器存取速度比栈快的多，dvm可以根据硬件实现最大的优化，比较适合移动设备。
- 3. class文件存在很多的冗余信息，dex工具会去除冗余信息，并把所有的class文件整合到dex文件中。减少了I/O操作，提高了类的查找速度

## 51. 内存泄漏

- 1. 其他线程持有一个Listener，Listener操作activity。那么在线程么有完毕的时候，activity关闭了，原本是要被回收的但是，不能被回收。
- 2. 例如Handler导致的内存泄漏，Handler就相当于Listener。
- 3. 在activity关闭的时候注意停止线程，或者将Listener的注册取消
- 3. 使用弱引用，这样即使Listener持有了activity，在GC的时候还是会回收
- 4. 工具：LeakCanary

## 2.43过度绘制、卡顿优化:

- 1.过度绘制:
  - 1.移除Window默认的Background:  
`getWindow.setBackgroundDrawable(null);`
  - 2.移除XML布局文件中非必需的Background
  - 3.减少布局嵌套(扁平化的一个体现，减少View数的深度，也就减少了View树的遍历时间，渲染的时候，前后期的工作，总是按View树结点来)
  - 4.在引入布局文件里面，最外层可以用merge替代LinearLayout,RelativeLayout，这样把子UI元素直接衔接在include位置
  - 5.工具：HierarchyViewer 查看视图层级
- 2.卡顿优化：16ms数据更新

## 2.44apk瘦身:

- 1.classes.dex：通过代码混淆，删掉不必要的jar包和代码实现该文件的优化
- 2.资源文件：通过Lint工具扫描代码中没有使用到的静态资源
- 3.图片资源：使用tinypng和webP，下面详细介绍图片资源优化的方案,矢量图
- 4.SO文件将不用的去掉，目前主流app一般只放一个arm的so包

## 2.45ANR的形成，各个组件上出现ARN的时间限制是多少

- 1.只要是主线程耗时的操作就会ARN 如io
- 2.broadcast超时时间为10秒 按键无响应的超时时间为5秒 前台service无响应的超时时间为20秒，后台service为200秒

## 2.46Serializable和Parcelable 的区别

- 1.P 消耗内存小
- 2.网络传输用S 程序内使用P
- 3.S将数据持久化方便
- 4.S使用了反射 容易触发垃圾回收 比较慢

## 2.47 SharedPreferences源码简述

- 1. 储存于硬盘上的xml键值对，数据多了会有性能问题
- 2. ContextImpl记录着SharedPreferences的重要数据，文件路径和实例的键值对
- 3. 在xml文件全部内加载到内存中之前，读取操作是阻塞的，在xml文件全部内加载到内存中之后，是直接读取内存中的数据
- 4. apply因为是异步的没有返回值, commit是同步的有返回值能知道修改是否提交成功
- 5. 多并发的提交commit时，需等待正在处理的commit数据更新到磁盘文件后才会继续往下执行，从而降低效率；而apply只是原子更新到内存，后调用apply函数会直接覆盖前面内存数据，从一定程度上提高很多效率。 3.edit()每次都是创建新的EditorImpl对象。
- 6. 博客推荐：[全面剖析SharedPreferences](#)

## 2.48 操作系统如何管理内存的：

- 1. 使用寄存器进行将进程地址和物理内存进行映射
- 2. 虚拟内存进行内存映射到硬盘上增大内存
- 3. 虚拟内存是进行内存分页管理
- 4. 页表实现分页，就是 页+地址偏移。
- 5. 如果程序的内存存在硬盘上，那么就需要用页置换算法来将其调入内存中：先进先出、最近未使用最少等等
- 6. 博客推荐：[现代操作系统部分章节笔记](#)

## 2.49 浏览器输入地址到返回结果发生了什么

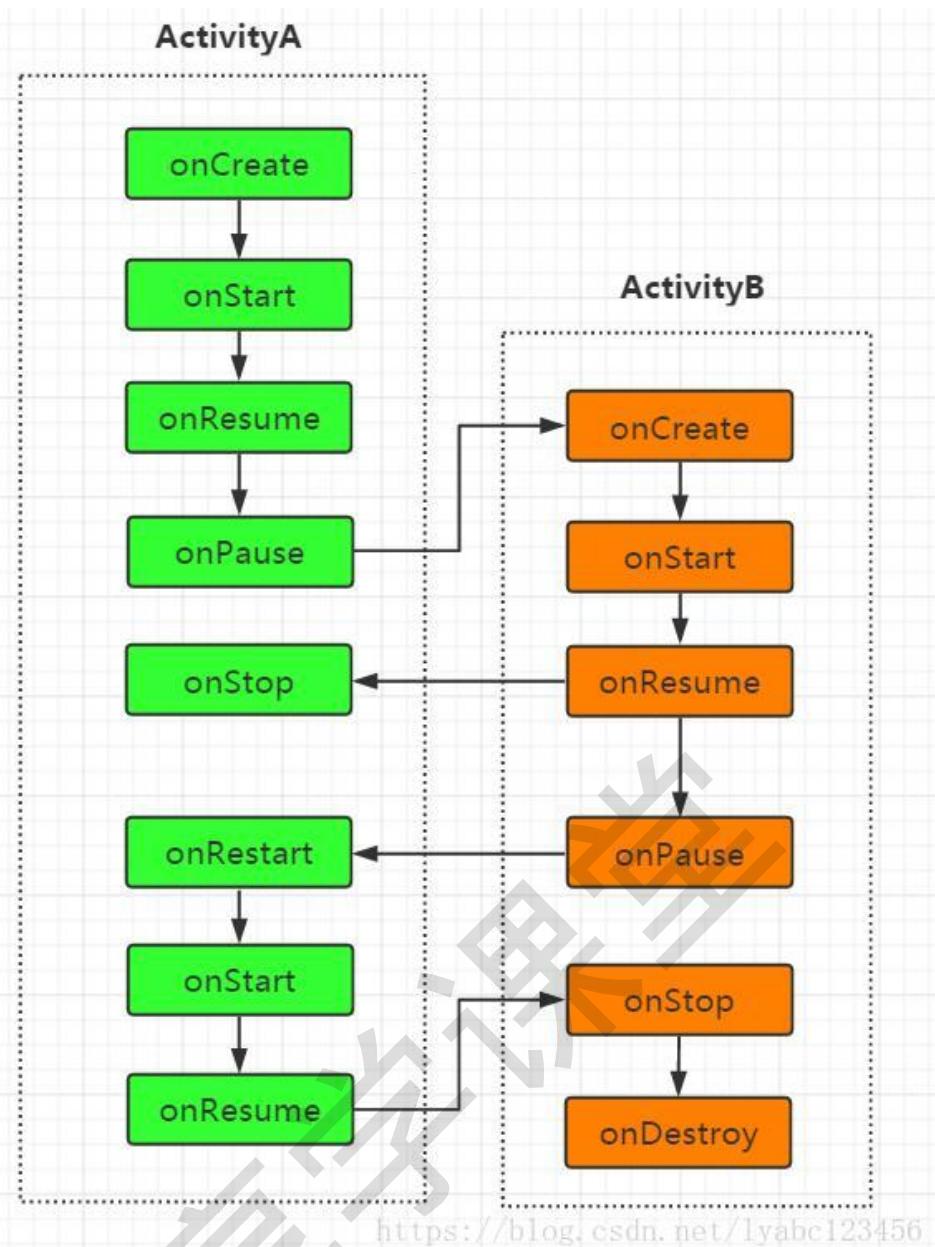
- 1. DNS解析
- 2. TCP连接
- 3. 发送HTTP请求
- 4. 服务器处理请求并返回HTTP报文
- 5. 浏览器解析渲染页面
- 6. 连接结束

## 59.java泛型类型擦除发生在什么时候，通配符有什么需要注意的。

- 1.发生在编译的时候
- 2.PECS， extends善于提供精确的对象 A是B的子集， Super善于插入精确的对象 A是B的超集
- 3.博客推荐：[Effective Java笔记（不含反序列化、并发、注解和枚举）](#)、[android阿里面试java基础锦集](#)

## 2.50activity的生命周期

- 1.a启动b，后退键再到a的生命周期流程：a.create->a.start->a.resume->a.pause->b.create->b.start->b.resume->b界面绘制->a.stop->b.pause->a.restart->a.start->a.resume->b.stop->b.destroy
- 2.意外销毁会调用saveInstanceState，重新恢复的时候回调用restoreInstanceState。储存数据的时候使用了委托机制，从activity->window->viewGroup->view 会递归调用save来保持本view的数据，restore则是递归恢复本view数据。我们可以在里面做一些自己需要的数据操作。



从图中可以得出两点总结：

- 当跳转到另一个Activity时，总是要等到另一个Activity的onResume方法执行完才会返回当前Activity的生命周期继续执行
- 当跳转到另一个Activity时，总是要等到当前的Activity的onPause方法执行完才会执行另一个Activity的生命周期（onCreate或onRestart）

## 2.51面试常考的算法

- 1.快排、堆排序为首的各种排序算法
- 2.链表的各种操作：判断成环、判断相交、合并链表、倒数K个节点、寻找成环节点
- 3.二叉树、红黑树、B树定义以及时间复杂度计算方式

- 4.动态规划、贪心算法、简单的图论
  - 5.推荐书籍：算法导论，将图论之前的例子写一遍
- 62.Launcher进程启动另外一个进程的过程：[启动一个app](#)

## 63.开源框架源码

- 1.Fresco
  - 1.mvc框架：
    - 1.Controller控制数据显示在Hierarchy中的Drawable的显隐
    - 2.ImagePipeline在Controller中负责进行数据获取，返回的数据是CloseableImage
    - 3.Drawee把除了初始化之外的操作全部交给Holder去做，Holder持有Controller和Hierarchy
  - 2.Drawable层次以及绘制：
    - 1.如果要绘制一次Drawable就调用invalidateSelf()来触发onDraw()
    - 2.Drawable分为：容器类(保存一些Drawable)、自我绘制类(进度条)、图形变换类(scale、rotate、矩阵变换)、动画类(内部不断刷新，进行webp和gif的帧绘制)
    - 3.ImagePipeline返回的CloseableImage是由一个个DrawableFactory解析成Drawable的
    - 4.webp和gif动画是由jni代码解析的，然后其他静态图片是根据不同的android平台使用BitmapFactory来解析的
  - 3.职责链模式：producer不做操作标n，表示只是提供一个consumer。获取图片→解码图片缓存Producer→后台线程Producer→client图片处理producer(n)→解码producer(n)→旋转或剪裁producer(n)→编码图片内存缓存producer→读硬盘缓存producer→写硬盘缓存producer(n)→网络producer提供CloseableImage《-解码图片缓存consumer《-client图片处理consumer《-解码consumer《-旋转或剪裁consumer《-编码图片内存缓存consumer《-写硬盘缓存consumer《-图片数据
  - 4.内存缓存：
    - 1.一个CountingLruMap保存已经没有被引用的缓存条目，一个CountingLruMap保存所有的条目包括没有引用的条目。每当缓

存策略改变和一定时间缓存配置的更新的时候，就会将待销毁条目Map中的条目一个个移除，直到缓存大小符合配置。

- 2.这里的引用计数是用Fresco组件实现的引用计数器。
- 3.缓存有一个代理类，用来追踪缓存的存取。
- 4.CountingLruMap是使用LinkedHashMap来储存数据的。
- 5.硬盘缓存：
  - 1.DefaultDiskStorage使用Lru策略。
  - 2.为了不让所有的文件集中在一个文件中，创建很多命名不同的文件夹，然后使用hash算法把缓存文件分散
  - 3.DiskStorageCache封装了DefaultDiskStorage，不仅进行缓存存取追踪，并且其在内存里面维持着一个键值对，因为文件修改频繁，所有只是定时刷新，因此如果在内存中找不到，还要去硬盘中找一次。
  - 4.删除硬盘的缓存只出现在硬盘数据大小超限的时候，此时同时也会删除缓存中的key，所以不会出现内存中有key，但是硬盘上没有的情况。
  - 5.在插入硬盘数据的时候，采用的是插入器的形式。返回一个Inserter，在Inserter.writeData()中传入一个CallBack(里面封装了客户端插入数据的逻辑和文件引用)，让内部实现调用CallBack的逻辑来插入文件数据，前面写的文件后缀是.temp,只有调用commit()之后才会修改后缀，让文件对客户端可见。
  - 6.使用了java提供的FileTreeVisitor来遍历文件
- 6.对象池：
  - 1.使用数组来存储一个桶，桶内部是一个Queue。数组下标是数据申请内存的byte大小，桶内部的Queue存的是内存块的。所以数组使用的是稀疏数组
  - 2.申请内存的方式有两种 1.java堆上开辟的内存 2.ashme 的本地内存中开辟的内存
- 7.设计模式：Builder、职责链、观察者、代理、组合、享元、适配器、装饰者、策略、生产者消费者、提供者
- 8.自定义计数引用：类似c++智能指针
  - 1.使用一个静态IdentityHashMap <储存需要被计数引用的对象，其被引用的次数>

- 2.用SharedReference分装需要被计数引用的对象，提供一个销毁资源的销毁器，提供一个静态工厂方法来复制自己，复制一个引用计数加一。提供一个方法销毁自己，表示自己需要变成无人引用的对象了，此时引用计数减一。
- 3.引用计数归零，销毁器将销毁资源，如bitmap的recycle或者是jni内存调用jni方法归还内存。
- 9.博客推荐：[Android Fresco源码文档翻译](#)、[从零开始撸一个Fresco之硬盘缓存](#)
- 2.oKhttp：
  - 1.同步和异步：
    - 1.异步使用了Dispatcher来将存储在 Deque 中的请求分派给线程池中各个线程执行。
    - 2.当任务执行完成后，无论是否有异常，finally代码段总会被执行，也就是会调用Dispatcher的finished函数，它将正在运行的任务Call从队列runningAsyncCalls中移除后，主动的把缓存队列向前走了一步。
  - 2.连接池：
    - 1.一个Connection封装了一个socket，ConnectionPool中储存着所有的Connection，StreamAllocation是引用计数的一个单位
    - 2.当一个请求获取一个Connection的时候要传入一个StreamAllocation，Connection中存着一个弱引用的StreamAllocation列表，每当上层应用引用一次Connection，StreamAllocation就会加一个。反之如果上层应用不使用了，就会删除一个。
    - .ConnectionPool中会有一个后台任务定时清理 StreamAllocation列表为空的Connection。5分钟时间，维持5个socket
  - 3.选择路线与建立连接
    - 1.选择路线有两种方式：
      - 1.无代理，那么在本地使用DNS查找到ip，注意结果是数组，即一个域名有多个IP，这就是自动重连的来源
      - 2.有代理HTTP：设置socket的ip为代理地址的ip，设置socket的端口为代理地址的端口

- 3.代理好处：HTTP代理会帮你在远程服务器进行DNS查询，可以减少DNS劫持。
- 2.建立连接
  - 1.连接池中已经存在连接，就从中取出(get)RealConnection，如果没有命中就进入下一步
  - 2.根据选择的路线(Route)，调用Platform.get().connectSocket选择当前平台Runtime下最好的socket库进行握手
  - 3.将建立成功的RealConnection放入(put)连接池缓存
  - 4.如果存在TLS，就根据SSL版本与证书进行安全握手
  - 5.构造HttpStream并维护刚刚的socket连接，管道建立完成
- 4.职责链模式：缓存、重试、建立连接等功能存在于拦截器中网络请求相关，主要是网络请求优化。网络请求的时候遇到的问题
- 5.博客推荐：[Android数据层架构的实现 上篇](#)、[Android数据层架构的实现 下篇](#)
- 3.okio
  - 1.简介：
    - 1.sink：自己-》别人
    - 2.source：别人-》自己
    - 3.BufferSink：有缓存区域的sink
    - 4.BufferSource：有缓存区域的source
    - 5.Buffer：实现了3、4的缓存区域，内部有Segment的双向链表，在转移数据的时候，只需要将指针转移指向就行
  - 2.比java io的好处：
    - 1.减少内存申请和数据拷贝
    - 2.类少，功能齐全，开发效率高
  - 3.内部实现：
    - 1.Buffer的Segment双向链表，减少数据拷贝
    - 2.Segment的内部byte数组的共享，减少数据拷贝
    - 3.SegmentPool的共享和回收Segment
    - 4.sink和source中被实际操作的其实是Buffer，Buffer可以充当sink和source

- 5.最终okio只是对java io的封装，所有操作都是基于java io 的

码农课堂